MASTER EN SCIENCES INFORMATIQUES À FINALITÉ SPÉCIALISÉE EN SOFTWARE ENGINEERING

L'influence des code smells sur l'impact environnemental du logiciel

De Boeck, Renaud

Award date:
2019

Awarding institution:
Universite de Namur

UNIVERSITÉ
DE NAMUR

FACULTÉ
D'INFORMATIQUE

L'influence des code smells sur l'impact environnemental du logiciel

Renaud De Boeck

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Université de Namur
Faculté d'informatique
Année académique 2018–2019

L'influence des code smells sur l'impact environnemental du logiciel

Renaud De Boeck

Maître de stage : Michaël Petit

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Michaël Petit

Co-promoteurs : Benoît Vanderose et Michaël Petit

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Les sujets concernant le climat n'ont jamais été aussi présents qu'aujourd'hui et la numérisation a sa part à jouer dans ce domaine. Chaque action, bien que minime, allant vers une réduction de l'impact écologique est à considérer. Ce mémoire a pour objectif de fournir une vue générale de l'influence que peut avoir le code source d'un logiciel mal conçu et de mauvaise qualité. Cette influence peut concerner des étapes du cycle de vie du logiciel telle que la maintenance mais peut également avoir un impact direct sur l'environnement. Pour remplir cet objectif, nous avons défini ce qui pouvait appauvrir la qualité d'un code source et nous en avons déterminé son impact sur base d'études existantes. Pour contrer cette influence négative sur l'environnement, il existe des techniques dites de refactoring qui permettent de rendre le code plus performant et de meilleure qualité. Sur base de cette matière, nous avons pu construire un modèle représentant cette influence pouvant servir de base théorique à la conception d'un logiciel capable d'estimer l'impact environnemental d'un code source.

Mots-Clés : *Green IT, Impact environnemental du code, Code Smells, Refactoring*

Abstract

Climate issues have never been more present than today, and digitization has its part to play in this area. Each action, although minimal, going towards a reduction of the ecological impact is to be considered. The purpose of this thesis is to provide an overview of the influence that source code can have on poorly designed software. This influence may concern stages of the software life cycle such as maintenance but may also have a direct impact on the environment. To fulfill this objective, we defined what could impoverish the quality of a source code and we determined its impact based on existing studies. To counter this negative influence on the environment, there are so-called refactoring techniques that make the code more efficient and better. Based on this material, we have been able to build a model representing this influence that can serve as a theoretical basis for the design of a software capable of estimating the environmental impact of a source code.

Mots-Clés : *Green IT, Environmental impact of code, Code Smells, Refactoring*

Remerciements

Je tiens premièrement à remercier mes deux promoteurs Mr Petit Michaël et Mr Vanderose Benoît qui ont tous deux acceptés de m'accompagner et de travailler avec moi pour cette dernière et grande étape qu'est le mémoire. Je les remercie particulièrement pour le choix du sujet et la liberté offerte quant à la direction à prendre. De plus, ils ont pu me conseiller de façon à ce que je puisse constamment avancer et rester dans la bonne voie.

Pour finir, j'aimerais également remercier ma mère pour son soutien durant cette dernière année d'étude et plus particulièrement la motivation qu'elle m'a donné lors de la rédaction de ce mémoire.

Table des matières

Chapitre 1

Introduction

Depuis quelques années, la question du climat est de plus en plus présente et prise au sérieux. Le numérique est un domaine qui évolue, s'accroît continuellement et est de plus en plus accessible dans le monde entier. Ce domaine a donc un impact de plus en plus conséquent sur l'environnement. Premièrement parce que la production de tous ces appareils nécessite l'extraction de ressources, polluant et épuisant les ressources finies que possède la planète. Et deuxièmement parce que ces appareils, une fois produits, sont alimentés en électricité pour pouvoir fonctionner et faire exécuter les logiciels.

Le cycle de vie d'un logiciel est composé de plusieurs phases par lesquelles il est impératif de passer. Ces phases sont les suivantes :

- *La phase d'ingénierie des exigences* qui consiste à définir et comprendre les besoins des futurs utilisateurs du logiciel.
- *La phase de conception* qui consiste à décrire plus formellement le fonctionnement du système afin de faciliter sa réalisation.
- *La phase de développement* qui consiste à implémenter et mettre en place le système sur base des besoins.
- *La phase d'utilisation* qui correspond à la phase où les utilisateurs utilisent le logiciel une fois déployé.
- *La phase de maintenance* qui consiste à maintenir et à faire évoluer le logiciel.

Chacune de ces phases engendre différents impacts environnementaux mais peuvent aussi impacter les phases suivantes. Par exemple, J. Taina [55] a identifié qu'une phase d'ingénierie des exigences aura un impact environnemental lié essentiellement à l'électricité et au chauffage nécessaire au bâtiment dans lequel le processus se déroule. Cependant, plus la phase sera complète et rigoureuse, plus l'impact de la phase de conception et de développement sera moindre. La phase de conception est importante et joue un grand rôle sur de futurs éventuels impacts énergétiques du logiciel. C'est

durant cette phase que des décisions sont prises sur la manière dont le logiciel sera implémenté. De mauvais choix concernant un algorithme, des structures de données utilisées ou des choix sur l'interface graphique auront un impact plus ou moins important. Cet impact peut concerner la phase de maintenance dans le cas où les développeurs se rendent compte des mauvais choix lors de la phase de conception mais aura également un impact direct sur les ressources utilisées par la machine. La phase de développement, tout comme les deux phases précédentes, possède elle aussi sa propre empreinte carbonne. Notamment en prenant en compte toutes les activités nécessaires de l'équipe de développement (les déplacements, les réunions, l'utilisation du chauffage, de l'électricité du bâtiment, ...) qui sont responsables d'émissions de CO2. Cette phase est également susceptible d'impacter la phase de maintenance. Un mauvais développement va nécessiter une phase de maintenance plus longue et plus complexe. Sachant que la phase de développement est basée sur la phase de conception, il est important que cette dernière ait bien été réalisée. Les deux dernières phases reposent toutes les deux sur les phases qui précèdent. La phase d'utilisation peut avoir un impact considérable sur la consommation énergétique de la machine dans le cas où le logiciel a mal été conçu. Un logiciel peu performant va solliciter davantage la machine en terme de ressources. Finalement, la phase de maintenance peut être plus ou moins conséquente en fonction de la réalisation de toutes les phases précédentes. Cette dernière phase aura des impacts similaires à la phase de développement puisqu'il s'agit de développement supplémentaire.

Ce mémoire traite principalement de l'impact environnemental du code source des programmes. C'est-à-dire que les phases concernant les exigences et la conception du logiciel ne seront pas prises en compte. Bien que cet impact du code source ait moins d'impact que la production et le recyclage des appareils (Ordinateurs, appareils mobiles, ...), il n'existe encore que très peu d'études sur le sujet. L'objectif est de comprendre la raison pour laquelle le code source d'un programme peut demander davantage de ressources à la machine lors de son exécution. Ces ressources peuvent être du temps de calcul ou de l'espace mémoire. Par exemple, plus un processeur sera sollicité, plus il consommera de l'énergie.

L'objectif de ce mémoire est donc dans un premier temps d'apporter une analyse des sources existantes concernant les mauvaises pratiques logicielles et le lien que celles-ci peuvent avoir avec l'impact environnemental du logiciel. Dans un second temps, ce mémoire vise à fournir un modèle d'influence synthétisant la connaissance obtenue sur les mauvaises pratiques et leur influence sur l'impact environnemental. Pour remplir cet objectif, les questions de recherches suivantes ont été posées :

— Quelles sont les mauvaises pratiques de programmation recensées dans la littérature ?

— Quel lien existe-t-il entre ces mauvaises pratiques et l'impact environnemental du logiciel dans les phase d'utilisation et de maintenance ?
— Comment détecter et corriger les mauvaises pratiques logicielles ?
— À quel point peut-on considérer la correction des mauvaises pratiques dans le but d'avoir un gain sur l'impact environnemental du logiciel ?

Le mémoire est structuré de la façon suivante : le chapitre 2 introduit de manière globale l'impact du numérique et l'impact même du code source d'un programme.

Nous verrons ensuite au chapitre 3 qu'il existe de nombreuses mauvaises pratiques (que l'on nommera *code smells*) qui influencent la qualité du code et nous différencierons les programmes executés par des ordinateurs de bureau (ordinateurs fixes et ordinateurs portables) de ceux exécutés par des appareils mobiles (smartphones, tablettes, ...). Ce même chapitre présente différents outils existants qui permettent de détecter ces mauvaises pratiques ainsi que la comparaison de ces outils.

Nous verrons ensuite, au chapitre 4, l'impact environnemental que peuvent avoir les codes sources de moins bonne qualité provenant d'une part, des applications de bureau et de l'autre, des applications mobiles.

Le chapitre 5 propose dès lors des techniques de refactoring permettant de corriger les erreurs de conception et les mauvaises pratiques qui causent un impact environnemental plus important et qui ont donc pour objectif d'augmenter la qualité du code de façon à limiter cet impact.

Pour finir, le chapitre 6 propose une synthèse générale faisant le lien entre les code smells, leur impact et les techniques de refactoring associées. Dont un modèle permettra d'obtenir une meilleure visualisation et pourra servir de base théorique à la conception d'un outil d'analyse de code permettant d'en évaluer l'aspect environnemental.

Chapitre 2

Impact environnemental du numérique

2.1 L'impact environnental global du numérique

Aujourd'hui c'est bien connu, le numérique est de plus en plus présent, de plus en plus accessible et ne cesse de s'étendre. Le rapport Clicking Clean, publié par GreenPeace en 2017 [1] indique que la consommation mondiale d'électricité du numérique est estimée à 7%. Tandis que le nombre d'internautes s'élevait déjà approximativement à 4 millards en 2017, celui-ci pourrait atteindre les 5 millards d'ici 2020. Ce qui sous-entend une croissance considérable en terme d'utilisation d'appareils connectés et donc une augmentation inévitable de la consommation d'électricité dans ce domaine.

L'impact environnemental du numérique représente aujourd'hui et au niveau mondial près de 4% des émissions de gaz à effet de serre [2]. Et la consommation énergétique n'est pas la seule responsable. La production des appareils numériques nécessite des matières premières (matériaux rares, eau douce, ressources fossiles, ...) qui s'épuisent d'années en années et dont le recyclage est difficile et engendre des déchets pouvant impacter la biodiversité [3]. Ces appareils et machines connectés sont utilisés pour accéder à l'information, utiliser un service, travailler, se divertir ou bien d'autres choses encore...Tout cela requiert l'existence de programmes, d'applications ou de sites web.

Dans ce mémoire, pour rappel, l'aspect écologique du numérique sera essentiellement focalisé sur le code source et l'utilisation de ces programmes dans le contexte des applications standards d'entreprise et des applications mobiles. La section suivante introduit ce que le code source d'un tel programme peut avoir comme impact sur l'environnement et c'est sur cet aspect-là uniquement que les chapitres suivants s'appuieront.

2.2 L'impact environnemental du code source d'un programme

Le code source d'un programme [1] est globablement définit par l'ensemble des fichiers textes pouvant être écrits dans divers langages de programmation et contenant donc toutes les instructions nécessaires à son exécution. Les langages de programmation sont humainement lisibles et compréhensibles. Cependant, ce programme va devoir être compilé de façon à ce qu'il puisse être interprété par le processeur [2] de la machine sur lequel le programme s'exécute.

Bien que l'écriture de lignes de code n'a que très peu d'impact en tant que tel sur l'environnement [5], le développeur doit néanmoins bien réfléchir et prendre du recul concernant la conception de son programmme. Des fonctionnalités inutilisées, l'utilisation excessive de frameworks, un mauvais choix d'algorithmes, et toutes mauvaises pratiques diminuant la qualité du code sont des éléments qui contribueront à une demande plus élevée en puissance de calcul, en mémoire etc. pour la machine. Ce qui, comme nous le verrons dans un chapitre ultérieur, a naturellement un impact plus élevé sur l'environnement.

Par exemple, entre 2010 et 2015, le poids des pages web a été multiplié par 3 alors que les sites web offrent toujours le même service [5]. Les développeurs se servent des nombreux frameworks existants, rendant les pages web bien plus lourdes qu'auparavant. L'emploi d'un framework n'est pas une mauvaise pratique en soi, mais bien souvent, ceux-ci sont importés dans leur intégralité alors que l'on ne se sert uniquement que d'une partie. De même que pour le choix des algorithmes qui impacte directement la puissance de calcul nécessaire. Un exemple très intéressant est celui donné par Frédéric Bordage, le fondateur du site GreenIT.fr expliquant que la société IBM [3] a divisé par 100 son nombre de serveurs à Zurich simplement en modifiant leur algorithme d'ordonnancement des tâches [5].

Le fait est donc ici, que plus le code source d'un programme est lourd, complexe et demandeur de ressources, plus celui-ci demande une puissance de calcul élevée ou un espace mémoire plus important et donc une consommation énergétique plus importante.

Il est également important de noter que comme l'indique la section précédente, l'accès aux applications est de plus en plus grandissant. Une augmentation de l'efficacité énergétique de ces dernières, à l'échelle mondiale,

1. Un programme est ici considéré comme étant un programme classique, une application standard, mobile ou web.

2. Il s'agit de l'unité centrale de calcul de la machine. Ce composant permet d'exécuter les instructions machine.

3. International Business Machines Corporation, connue sous le sigle IBM, est une société multinationale américaine présente dans les domaines du matériel informatique, du logiciel et des services informatiques [6].

pourrait alors entraîner des changements majeurs et positifs sur la consommation énergétique dans le domaine du numérique [40]. Si l'on considère une application qui est utilisée par des millions d'utilisateurs, il faut garder à l'esprit qu'une méthode qui a un temps d'exécution de quelques millisecondes (ms) et qui est exécutée une fois pour chaque utilisateur, ce temps d'exécution est au final systématiquement additionné. Par exemple, une méthode avec un temps d'exécution de 10 ms qui est exécutée un million de fois, représente 10 millions de ms soit 166,67 secondes représentant 2,78 heures d'exécution par la machine. C'est pourquoi de simples optimisations sur les algorithmes et de manière plus globale, sur la conception de l'application peut considérablement réduire le temps d'exécution total des applications et ainsi réduire la consommation d'énergie liée à cette exécution.

Un autre point pertinent concernant le code source des programmes mais qui ne sera pas traité dans ce travail est le choix du langage de programmation. Globalement, il est possible de réaliser la même chose avec la majorité des langages de programmation existants mais ne sont cependant pas tous aussi performants les uns des autres et ne se valent pas sur le plan énergétique. En effet, une étude portugaise [50] a classé les différents langages selon leur rapidité d'exécution, leur consommation énergétique et leur utilisation de la mémoire. Montrant ainsi que des langages comme le C, C++ et le Rust sont des langages très peu énergivores tandis que des langages comme Ruby, Python et Perl le sont fortement comme indiqué dans le tableau 2.1 où l'on constate que le langage Python est 75 fois plus énergivore que le langage C. Le chapitre suivant concerne l'analyse statique de ce code

Langage	Énergie
C	1.00
Rust	1.03
C++	1.43
...	
Java	1.98
...	
C#	3.14
...	
PHP	29.30
...	
Ruby	69.91
Python	75.88
Perl	79.58

TABLE 2.1 – Langages et leur consommation d'énergie [50]

source où une liste exhaustive des mauvaises pratiques pouvant avoir un impact sur la consommation énergétique de la machine est répertoriée.

Chapitre 3

Code Smells : mauvaises pratiques logicielles

Dans la section 3.1 dans ce chapitre, nous verrons toutes les mauvaises pratiques logicielles qui peuvent avoir un lien avec l'impact environnemental du logiciel. Dans la section 3.2, nous verrons les mesures logicielles qui peuvent considérablement impacter la qualité du code et son exécution. Et la section 3.3 clôturera le chapitre en présentant les différents outils d'analyse permettant de détecter les code smells.
Le chapitre 4 expliquera l'impact environnemental que peut engendrer un code mal conçu avec la présence de ces code smells.

3.1 Code Smells

Les "Code Smells" (ou mauvaises odeurs de code en français) [7] sont les mauvaises pratiques de conception logicielle que l'on peut retrouver dans le code source d'un programme. Il en existe sous différentes catégories que nous allons parcourir dans cette section. Nous différencierons les code smells des applications standards (de bureau), que l'on appelle également *code smells de Fowler* des code smells spécifiques aux applications mobiles et plus particulièrement à Android. Les sous-sections 3.1.1 à 3.1.6 font références aux code smells de Fowler et la sous-section 3.1.7 fait référence aux code smells spécifiques à Android.

3.1.1 Les bloaters

Les bloaters, ou ballonements / gonflements en français sont une catégorie de Code Smells qui représente du code, des méthodes ou des classes qui, durant leur évolution, leur taille est devenue tellement imposante, qu'il est devenu complexe de les maintenir efficacement [8, 10].

Méthodes longues / Long Methods

Une mauvaise pratique pour les développeurs est de rendre les méthodes bien trop longues et trop complexes. Les avis concernant la taille adéquate d'une méthode peuvent légèrement différer bien qu'en moyenne, il resort qu'une bonne méthode contient entre 4 et 20 lignes de code. Au delà, il devient préférable de la décomposer [7, 9]. Fowler Martin et Beck Kent [11] expliquent dans leur description pour ce "smell" qu'il y a plusieurs bonnes raisons d'avoir des méthodes courtes. La raison principale est de partager la logique. En décomposant une longue méthode, il est souvent aisé de partager sa logique en plusieurs petites méthodes possèdant chacune sa propre responsabilité, avec un nom approprié. D'autant plus qu'une longue méthode peut très bien contenir du code dupliqué. Les auteurs précédemment cités décrivent également pourquoi le code d'une petite méthode est bien plus facile à comprendre. Décomposer une longue méthode en plus petites méthodes permet donc une meilleure compréhension du code original, ne nécessite plus ou peu de commentaires puisque celles-ci se suffisent à elles-mêmes et pour finir, le système entier est alors lui-même plus compréhensible, plus facile à maintenir et contient moins de duplication [13].

La raison de ce problème est que les développeurs ont tendance à ajouter des instructions dans une méthode sans jamais en retirer.
Et c'est compréhensible dans le sens où mentalement, il est plus compliqué de créer une nouvelle méthode que d'ajouter quelques instructions dans une méthode déjà existante [8] :

"Ce n'est que deux lignes, il n'y a pas besoin de créer toute une nouvelle méthode pour cela..."

Ce qui signifie au final qu'une ligne de code est ajoutée à chaque fois jusqu'à ce que l'on se retrouve avec un code dans une méthode qui est difficile à lire et à comprendre...Le problème peut également être dû à un trop grand nombre de paramètres internes, de boucles, de conditions ou de valeurs de retour au sein de la méthode.

Classes larges / Large Class et God Class

Dans le même genre que les méthodes longues, les classes larges est une pratique qui décrit des classes contenant beaucoup d'attributs, de méthodes et de lignes de code [8] indiquant bien souvent qu'elle possède plus de responsabilités qu'elle ne devrait en avoir [11, 13]. La raison du problème, tout comme pour les méthodes longues, est qu'une classe commence toujours en étant petite et au fur et à mesure où le programme évolue, les classes deviennent de plus en plus grosses. À nouveau, les développeurs pensent que c'est moins taxant d'ajouter une nouvelle fonctionnalité dans une classe existante plutôt que d'en créer une nouvelle dédiée à cette fonctionnalité [8].

Une bonne pratique est de conserver le principe de responsabilité unique pour chaque classe. Si une phrase ne suffit pas à décrire ce qu'elle fait, c'est qu'il y a de fortes chances qu'elle ait plusieurs responsabilités et qu'elle en fait beaucoup trop.

Un code smell similaire aux classes larges est la *God Class* qui correspond à une classe qui connait ou fait beaucoup trop de choses. Le problème est similaire aux classes larges puisque les God Class sont énormes en terme de ligne de code avec un couplage bien trop important.

Obsession des primitives / Primitive Obsession

L'"obsession des primitives" n'est pas un gonflement de code (bloater) en soi mais est une mauvaise pratique qui en est la cause. Les primitives sont des types de données génériques et de base que l'on retrouve dans la plupart des langages de programmation. Il s'agit notamment des types tels que les entiers, les chaînes de caractères, les décimaux (float/double), les tableaux, etc. Tandis que les classes sont des types de données bien plus spécifiques en fonction de ce dont on a besoin [13].

Cette mauvaise pratique apparaît lorsque l'on commence à utiliser des types primitifs un peu partout dans le code. En utilisant par exemple des entiers pour représenter un numéro de téléphone ou des chaînes de caractères pour représenter le symbole d'une monnaie.

Si un type de données est suffisamment complexe, il est préférable de créer une classe pour le représenter. De plus, les classes permettent une représentation des choses plus simple et plus élégante que les types primitifs. Voici un exemple très simple de classe contenant une "obsession des primitives" :

```python
class User:

    def __init__(self):
        self.firstname = "Renaud"
        self.lastname = "De Boeck"
        self.phone = "+32 472 00 00 00"
        self.street_name = "Rue du Paradis"
        self.street_num = "16"
        self.zip_code = "1315"
        self.country = "BE"
        #...
```

FIGURE 3.1 – Classe représentant le smell *Obsession des primitives*.

La classe User représentée à la Figure 3.1 peut ainsi contenir une longue série d'attributs. Ces attributs sont utilisés dans les différentes méthodes et peuvent engendrer de la duplication de code et de manière générale, une complexité plus importante à la classe, la rendant moins compréhensible.

La raison qui provoque ce problème est similaire aux deux précédents, il est aisé d'ajouter un simple attribut pour représenter une nouvelle donnée sans penser au fait qu'avec le temps, la classe en question va devenir de moins en moins maintenable [8]. Le chapitre 5 introduit des moyens et des méthodes de refactoring pour corriger et modifier ces mauvaises pratiques mais brièvement, dans ce cas-ci, céer une classe *Address* permettrait de représenter l'adresse de l'utilisateur en y regroupant les attributs appropriés.

Longues listes de paramètres / Long Parameter List

Comme son nom l'indique, avoir des fonctions ou méthodes avec un nombre important de paramètres est souvent signe que la conception de celle-ci n'est pas bonne. Cela empêche une bonne lisibilité et compréhension de la méthode et complique également l'utilisation et la refactorisation de celle-ci [11].

Il est courant de voir la liste des paramètres d'une méthode grandir lorsque celle-ci nécessite de nouvelles données. Le développeur faisant appel à cette méthode se verra prendre du temps à renseigner tous les paramètres et le code résultant ne sera pas forcément très élégant. Il est tout à fait possible de passer en paramètre directement un objet de classe plutôt que d'y renseigner une multitude de variables/attributs.

Agrégats de données / Data Clumps

Les agrégats de données sont des groupes de variables, souvent de type primitif, qui sont manipulés ensemble dans le programme [11]. Un exemple pour illustrer ce problème serait par exemple l'utilisation de trois variables de type *float* pour représenter une position dans un environnement 3D :

```
float x ; float y ; float z ;
```

Un bon moyen pour un développeur de se rendre compte si ce groupe de données est un "agrégat de données" est de simplement supprimer l'une des variables et déterminer si les variables restantes ont du sens ou pas. Dans notre exemple, si l'on retire l'une des variables, la position que l'on souhaite représenter n'a plus de sens puisque pour que ce soit le cas, nous avons besoin de trois données pour chacun des axes de l'environnement 3D. Il est donc dans ce cas, préférable de modifier cela en regroupant les variables dans une classe à part entière.

```
Position maPosition = new Position(x, y, z) ;
```

3.1.2 Les abuseurs orientés objet

Ce type de mauvaises pratiques représente le cas où un programme ou un code n'exploite pas entièrement les possibilités des principes de la pro-

grammation orientée-objet. Les code smells qui suivent ne sont à considérer uniquement qu'en cas de programmation orientée-objet.

Opérateur Switch / Switch Statements

L'instruction ou l'opérateur *Switch* est une bonne pratique dans la programmation procédurale [1] mais doit parfois être évitée dans la programmation orientée objet. Notamment dû à ces trois faits :

- **La violation du principe ouvert/fermé**, puisqu'à chaque fois que l'on voudra ajouter un nouveau *case*, il faudra modifier du code existant.
- **La difficulté à maintenir le code** puisqu'un Switch case est toujours contenu dans une seule fonction, ce qui peut la faire grandir à chaque nouveau besoin ce qui rend la maintenance de plus en plus compliquée.
- **Peut engendrer une duplication de code** comme le montre l'image ci-dessous.

```
switch ($em) {
    case A:
        doSomething("a");
        break;
    case B:
        doSomething("b");
        break;
    default:
        doSomething();
        break;
}
```

FIGURE 3.2 – Code redondant dans un switch case.

L'opérateur switch peut être remplacé en utilisant le polymorphisme. À noter donc que switcher sur un type d'objet n'est pas une bonne pratique mais switcher sur une valeur peut tout à fait être correct.

Champs temporaires / Temporary Field

Les champs temporaires sont les attributs d'une classe qui sont, par exemple, utilisés uniquement dans une seule méthode de la classe. Ce qui signifie que la plupart du temps, ces attributs sont vides ou contiennent des valeurs non pertinentes. Selon Mäntylä, M. V. et Lassenius, C. [10], cette mauvaise pratique va à l'encontre du principe d'encapsulation puisqu'au

1. Basée sur le concept d'appel procédurale, de fonctions.

final, ces attributs devraient être uniquement au sein de la méthode qui les utilise. La figure 3.3 est un exemple qui illustre ce code smell. On remarque bien que seule la méthode **method2()** utilise les attributs **self.amount** et **self.coef**. Il est dès lors recommendé de déplacer ces deux attributs au sein la méthode en question.

```
class MyClass:

    def __init__(self):
        self.amount = 2
        self.coef = 0.5
        #...

    def method1(self):
        #doSomething
        #Without using self.amount & self.coef

    def method2(self, value):
        return value * (self.amount * self.coef)

    #... Other methods
```

FIGURE 3.3 – Représentation du code smell “Champs temporaires”.

De plus, ces attributs devront être stockés en mémoire durant toute la durée de vie de l’objet instancié. Ce qui aura forcément un impact, comme nous le verrons au chapitre suivant.

Legs refusés / Refused Bequest

Le “leg refusé” est le fait d’avoir une classe qui hérite d’une autre classe mais où celle-ci n’utilise jamais voir que très peu les attributs et méthodes de la classe mère. Dans ce cas-là, l’héritage n’a pas lieu d’être et amène à avoir un code déroutant et peu élégant [11].

Classes alternatives avec différentes interfaces / Alternative Classes with Different Interfaces

Des classes alternatives avec différentes interfaces sont tout simplement l’utilisation de deux classes ou fonctions ayant des noms différents mais qui sont néanmoins similaires au point d’avoir la même fonction/responsabilité au sein du programme.

Cette mauvaise pratique peut arriver lorsque le développeur qui crée cette classe n’a pas connaissance de l’existence de la classe ou fonction similaire déjà existante. Cela ajoute du code redondant qui nécessite de la maintenance supplémentaire et inutile.

3.1.3 Les modifications préventives

Les modifications préventives sont une catégorie de mauvaises pratiques où, lorsqu'une modification doit être effectuée, cela engendre des modifications ailleurs dans le code source. Selon une règle suggérée par M. Fowler et Beck [11], il doit y avoir une relation "One-to-One" entre une classe et une modification éventuelle. Par exemple, si une modification est effectuée dans la base de données, cela ne doit affecter qu'une seule classe.

Changement divergent / Divergent Change

Le cas d'un "changement divergent" survient lorsque le développeur se retrouve à effectuer des modifications dans différentes méthodes de la même classe et n'ayant aucun lien direct entre-elles alors qu'une seule modification doit être apportée. Par exemple, si en ajoutant un nouveau type de produit dans le programme, il est nécessaire de modifier les méthodes concernant la recherche, l'affichage ainsi que le tri des produits [8], cela signifie qu'on se trouve dans un cas de changement divergent et qu'un problème de conception est bien présent au sein de la classe en question.

Chirurgie au fusil à pompe / Shotgun Surgery

Cette mauvaise pratique est le parfait inverse du smell précédent "*changement divergent*". Alors que ce dernier consiste à effectuer plusieurs modifications au sein de la même classe pour un seul changement initialement prévu, le *Shotgun Surgery* consiste à effectuer un seul changement mais dans différentes classes simultanément [8]. Cela est provoqué lorsque la responsabilité d'une fonctionnalité a été divisée et intégrée dans plusieurs classes.

Hiérarchies d'héritage parallèles / Parallel Inheritance Hierarchies

Une hiérarchie d'héritage parrallèle est un cas particulier du "Shotgun Surgery" smell qui consiste qu'à chaque fois qu'une sous-classe est créée pour une classe, une autre sous-classe doit être créée pour une autre classe. Cela est représenté à la Figure 3.4. Cette pratique va fatalement mener à de la duplication de code et à une organisation de code de moins en moins convenable.

3.1.4 Les dispensables

Un "dispensable" est un code ou une classe qui n'est tout simplement pas nécessaire. Sa présence peut rendre le code source moins propre, moins efficace et moins facile à comprendre. Cette section répertorie six mauvaises pratiques illustrant cette catégorie.

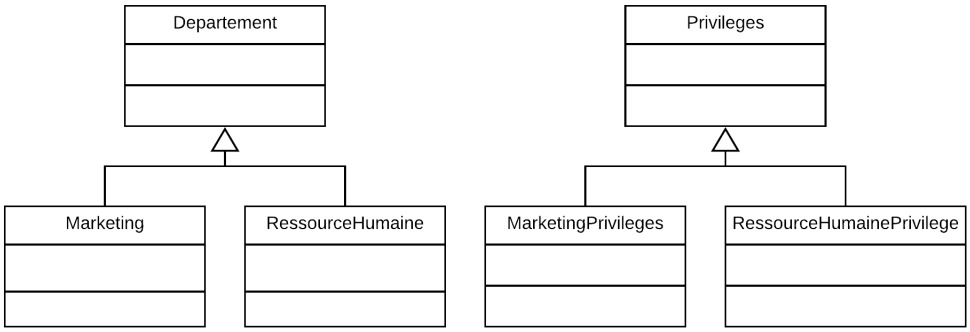


FIGURE 3.4 – Représentation du smell *hiérarchie d'héritage parallèle*.

Commentaires / Comments

Commenter son code source part d'une bonne intention. En revanche, un code largement commenté peut également faire référence à un code pas si évident ni intuitif. Ce smell est parfois appelé *déodorant* parce qu'il masque "*l'odeur*" d'un code qui peut amélioré. "Le meilleur des commentaires est un bon nom de méthode ou de classe" [8].

Mohamed Aladdin [9] confirme également qu'une mauvaise utilisation des commentaires est belle et bien une mauvaise pratique et donne pour conseils de :

— retirer les commentaires inutiles.
— si le code est évident, il n'est pas nécessaire de le commenter.
— Ne pas laisser du code ancien commenté dans le code source.

M. Fowler [11] quant à lui, conseille aux développeurs qu'avant de commenter son code, il faut d'abord essayer de le refactorer de façon à ce que le commentaire devienne superflu et n'est alors plus nécessaire.

Code dupliqué / Duplicated Code

La duplication de code est une mauvaise pratique classique qui consiste à copier-coller du code à différents endroits du programme. Il s'agit dès lors d'une duplication de code explicite puisque les deux portions de code sont identiques. À l'inverse, une duplication de code subtile représente deux portions de code non-identiques mais réalisant finalement la même chose.

La duplication de code, de manière générale, mène à une maintenance de code de plus en plus longue et complexe.

Classe paresseuse / Lazy Class

Une classe paresseuse ou “Lazy Class” en anglais, est une classe présente dans le projet mais qui est trop petite par rapport à ce qu'elle fait et ce qu'elle coûte en terme de maintenance. À savoir que l'ajout d'une classe dans un projet ajoute de la complexité à celui-ci. Soit la responsabilité de la classe doit augmenter, soit la classe tout entière doit disparaître.

Dans la catégorie des Bloaters (cf. Section 3.1.1), nous avons vu que certaines solutions préconisent de créer une nouvelle classe lorsqu'un ajout est nécessaire. En revanche, cette création doit être justifiée et utile afin d'éviter que la correction d'un code smell n'engendre l'apparition d'autres code smells. Comme par exemple, l'apparition d'une petite classe qui ne fait pratiquement rien (Lazy Class) et qui a été créée pour corriger un léger agrégats de données. De ce fait, la maintenabilité du programme peut avoir été améliorée tandis que sa complexité a augmenté.

Classe de données / Data Class

Les classes de données quant à elles, sont des classes qui contiennent uniquement des attributs et des méthodes pour accéder à ces attributs (getters et setters). Ce type de classes ne contient rien d'autre et sert de conteneur de données pour les autres classes. Ce “smell” rejoint l'idée des “Lazy class” où leur responsabilité doit être augmentée ou bien l'entièreté de la classe doit être supprimée.

Une classe doit donc contenir des données ainsi que des méthodes qui opèrent sur ces mêmes données [11].

Code mort / Dead Code

Un code mort est toute variable, paramètre, attribut, méthode ou classe qui ne sont plus utilisés dans le programme. Cette mauvaise pratique survient lorsque les besoins ont changé ou qu'une correction a été faite dans le code et que personne n'a pris le temps de nettoyer celui-ci. Il s'agit donc de code qui n'est jamais exécuté ni exploité par le programme. D'une part, cela rajoute de la complexité au programme et d'autre part, la maintenance de celui-ci est rendue plus difficile. Premièrement parce qu'un développeur peut prendre du temps à comprendre ce que fait le code pour finalement se rendre compte que ce code n'est pas utilisé et puis cela peut soulever des questions ; *est-ce que ce code est en cours d'implémentation et est nécessaire pour une future fonctionnalité ? Est-ce qu'un bloc de code commenté est un code qui est buggé et doit être corrigé, ou est-il lent ? . . .*

Généralité spéculative / Speculative Generality

La généralité spéculative est légèrement similaire au code mort dans le sens où cette pratique consiste à anticiper un comportement futur dans le programme. Il s'agit donc de classes, de méthodes, d'attributs ou même de paramètres qui ne sont pas utilisés mais qui sont néanmoins présents de façon à répondre à un besoin futur. Le problème étant que ce besoin futur peut ne jamais être implémenté et le développeur se retrouve avec un code qui devient de plus en plus compliqué à comprendre et à maintenir.

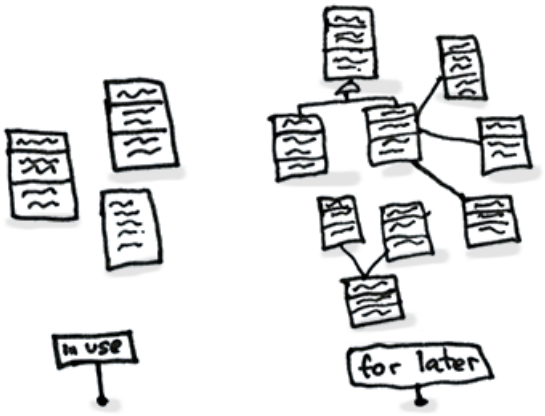


FIGURE 3.5 – Représentation du smell "Généralité spéculative" [8].

3.1.5 Les coupleurs

La catégorie des coupleurs regroupe les mauvaises pratiques qui mettent en oeuvre un couplage excessif entre les classes.

Fonctionnalité jalouse / Feature Envy

Une fonctionnalité jalouse (Feature Envy) est une méthode qui accède plus aux données d'une autre classe qu'à ses propres données. Dans ce cas, cette méthode est probablement dans la mauvaise classe. Ce problème arrive généralement lorsque des attributs sont déplacés dans une classe de données...[8]

Intimité inappropriée / Inappropriate Intimacy

L'intimité inappropriée (Inappropriate Intimacy) signifie que deux classes sont très liées entre-elles et passent beaucoup trop de temps à "fouiller" l'une dans l'autre. À l'inverse, il est de bonne pratique de faire en sorte que les classes se connaissent le moins possible afin de faciliter l'utilisation et la réutilisation de celles-ci.

Chaîne de messages / Message Chains

Une chaîne de message est une mauvaise pratique qui consiste à faire appel à une classe qui elle-même fait appel à une autre classe etc. Ce qui rend la classe de départ dépendante de la relation entre plusieurs classes qui n'ont pas forcément de liaison entre-elles [11].

Homme du milieu / Middle Man

L'homme du milieu (Middle Man) représente une classe qui ne fait qu'une seule action, qui est de déléguer du travail aux autres classes et ne contribue donc en rien au programme. La délégation en soi n'est pas une mauvaise pratique puisqu'il s'agit d'une fonctionnalité fondamentale en orienté-objet [11] mais dans ce cas-ci, la classe n'apporte aucune valeur supplémentaire, elle ne fait que passer le message aux autres classes. Le problème peut survenir lorsque la classe en question a été progressivement déplacée dans d'autres classes en ne lui laissant plus qu'une carapace vide ne faisant rien d'autre que déléguer [8].

3.1.6 Autres Smells

D'autres "smells" n'appartenant pas aux différentes catégories sont répertoriés dans cette sous-section.

Classe de librairie incomplète / Incomplete Library Class

Lorsqu'une librairie est importée et utilisée dans le proramme, il se peut que celle-ci ne réponde pas entièrement aux besoins du développeur. Par exemple, si une méthode d'une classe provenant de la librairie est inexistante ou non-implémentée mais est nécessaire pour le programme, on se retrouve dans un cas de classe de librairie incomplète. Et le fait est que la classe de la librairie n'est pas ou difficilement modifiable et donc la méthode en question va devoir être implémentée dans le programme alors qu'elle devrait se retrouver dans la librairie.

Étalement de solutions / Solution Sprawl

L'étalement de solutions (Solution Sprawl) est le fait d'implémenter plusieurs classes afin d'obtenir un résultat utile pour le programme. De même que pour du code qui réalise une action mais qui s'étale dans de nombreuses classes, cela est de l'étalement de solutions. Cela peut venir du fait qu'une fonctionnalité est rapidement ajoutée au programme sans avoir pris le temps de simplifier et de consolider sa conception [11].

3.1.7 Code smells spécifiques à Android

Dans le cadre du développement orienté vers des appareils mobiles, une étude [22] a indentifié quatre code smells qui ont directement un impact sur la consommation énergétique et sont donc essentiels à considérer dans le cadre de ce travail. Onze autres code smells [23], toujours spécifiques à Android et caractérisant un problème dans le code source sont également présentés dans cette sous-section.

Setter interne / Internal Setter

Dans le développement Android, l'utilisation de méthodes virtuelles telles que les getters et les setters sont assez coûteux et il est donc recommandé d'accéder directement aux attributs internes [23]. Bien que les compilateurs, depuis la version 2.3 d'Android optimisent automatiquement les getters ne retournant que l'attribut, ce n'est pas le cas pour les setters. Cela mène à des appels de fonctions additionnels, rendant l'application mobile moins performante, ce qui a un impact sur l'efficacité énergétique de celle-ci. Il est donc préférable d'utiliser

myAttribute = 'itsValue';
plutôt que
setMyAttribute('itsValue');

Thread qui fuit / Leaking Thread

Dans le développement Android, un Thread est une racine de Garbage Collector (GC). Le GC ne collectant pas les objets racines, si un Thread n'est pas correctement arrêté, celui-ci restera en mémoire durant toute l'exécution de l'application, causant dès lors une utilisation abusive de la mémoire.

Méthode ignorant les membres / Member Ignoring Method

Les méthodes ignorant les membres (Member Ignoring Method) (sous-entendu les attributs et autres méthodes de la classe) sont des méthodes non-statiques qui n'accèdent ou n'utilisent aucun membre interne de la classe. Ces méthodes devraient donc être renseignées comme statiques afin d'augmenter leur efficacité.

Boucle lente / Slow Loop

Les boucles lentes (Slow loop) sont des utilisations de l'opérateur *for* qui s'avère être plus lent que l'opérateur *for-each*. L'utilisation de ce dernier, à la place du simple *for* permet d'augmenter l'efficacité de l'application. Il peut aussi être recommandé d'optimiser les boucles lorsque cela est possible. Par exemple en l'interrompant à l'aide du mot-clef *break* si l'élément de recherche

est trouvé ou que l'opération à effectuer au sein de la boucle est terminée et qu'il reste des itérations.

Transmissions de données sans compression / Data Transmission Without Compression

Les données transmises sans compression est un code smell qui se produit lorsqu'un fichier est envoyé sur une infrastructure réseau sans que celui-ci ne soit compressé. Pouvant alors causer une surcharge de communication.

Release débogable / Debuggable Release

Une release [2] débogable signifie que la valeur de l'attribut *android :debuggable* provenant du fichier *AndroidManifest* vaut *true* alors que l'application a été mise en production. Cela représente un risque majeur au niveau de la sécurité puisque toutes les applications externes peuvent accéder au code source. En revanche, bien que ce code smell reflète un problème de sécurité, celui-ci n'a aucun impact quel qu'il soit sur la maintenabilité ou l'environnement. Il n'est dès lors aucunement pertinent pour ce travail.

Wakelock durable / Durable Wakelock

Un Wakelock est un mécanisme permettant à une application de garder l'appareil allumé de façon à pouvoir exécuter ses tâches. Une fois les tâches complétées, le verrou peut être relâché afin de diminuer la consommation de la batterie. Dans le cas où une méthode acquiert le verrou depuis une instance de la classe *PowerManager.WakeLock* d'Android sans jamais le relâcher, on peut considérer le code smell *Durable Wakelock* (DW). Ce problème cause inévitablement une consommation énergétique plus importante.

Format de données et Parseur inefficace / Inefficient Data Format and Parser

L'utilisation du *TreeParser* dans Android pour analyser des fichiers XML ou JSON est un exemple de format de données qui n'est pas suffisamment efficace et ralenti l'application. Utiliser des formats de données et des parseurs efficaces permettent une meilleure performance et rend l'application plus rapide et moins consommatrice de ressources.

Structure de données inefficace / Inefficient Data Structure

L'utilisation de structures de données inefficaces peuvent elles aussi ralentir l'application. Par exemple, le mappage d'un entier avec un objet grâce

2. Signifie une version de l'application mise en production.

aux HashMap : *HashMap<Integer, Object>* est une opération lente. Un choix de structure données plus efficace permet d'éviter de ralentir l'application.

Requête SQL inefficace / Inefficient SQL Query

Dans Android, l'utilisation de requêtes SQL est déconseillée car elle introduit une surcharge. Ce code smell est donc détecté lorsqu'une connexion JDBC est définie et que des requêtes SQL sont envoyées à un serveur distant. Bien évidemment, il existe d'autres solutions, qui seront présentées dans le chapitre sur le refactoring des code smells.

Fuite de classe interne / Leaking Inner Class

Une *classe interne* est une classe qui est définie dans une autre classe. Les fuites de classes internes (LIC) est un code smell définit par Reimann et al [32] comme étant une classe interne non statique contenant une référence à la classe externe. Ce qui peut avoir comme conséquence d'engendrer des fuites de mémoire. Cette pratique est souvent utilisée par les développeurs Android afin de gagner du temps sans porter attention à l'effet que cela peut avoir sur les performances de la mémoire.

Le code représenté par la Figure 3.6 est un exemple de Leaking Inner Class où une simple **AsyncTask** est créée et exécutée lorsque l'*Activity* est démarrée. Comme la classe interne doit avoir accès à la classe externe, des fuites de mémoire se produisent chaque fois que l'activité est détruite alors que l'**AsyncTask** est toujours en cours d'exécution.

Aucun résolveur de mémoire faible / No Low Memory Resolver

Il est tout à fait possible, pour un développeur Android de définir le comportement que peut avoir une application lorsque celle-ci s'exécute en arrière plan en utilisant la méthode *Activity.onLowMemory*. Cette méthode doit être utilisée pour nettoyer les caches et les ressources inutiles et sans elle, l'application peut mener à une utilisation de la mémoire anormale.

Données publiques / Public Data

Le code smell des données publiques survient lorsque des données privées sont stockées dans un *store* (telles que les préférences partagées ou les stockages internes) qui est publiquement accessible par les autres applications. C'est une mauvaise pratique peu intéressante dans le cadre de ce travail puisque l'impact ne touche que la sécurité de l'application et non sa performance ou son impact énergétique.

```java
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        new MyAsyncTask().execute();
    }

    private class MyAsyncTask extends AsyncTask {
        @Override
        protected Object doInBackground(Object[] params) {
            return doSomeStuff();
        }
        private Object doSomeStuff() {
            //do something to get result
            return new MyObject();
        }
    }
}
```

FIGURE 3.6 – Classe interne non-statique pouvant provoquer une fuite de mémoire. (Source : [33])

Gestionnaire d'alarme rigide / Rigid Alarm Manager

Android permet de gérer un système de gestion d'alarme à l'aide de la classe *AlarmManager* et qui permet d'exécuter des opérations à des moments spécifiques qui, une fois déclenchée, réveille l'appareil. Cela peut impacter l'efficacité énergétique et l'efficacité de la mémoire de l'application. On appelle le code smell *gestionnaire d'alarme rigide* pour les applications qui contiennent une instance de la classe *AlarmManager* pour laquelle l'alarme est réglée précisément à l'aide de la méthode *setRepeating()*. Ce qui peut être très drainant pour la batterie.

Fermable non fermé / Unclosed Closable

Selon la documentation officielle d'Android [34], une interface *Closeable* est une source ou une destination de données qui peut être fermée. La méthode *close* est donc appelée pour relâcher les resources que l'objet détient (tels que des fichiers ouverts par exemple). Un **Closeable non fermé** (Unclosed Closable) est l'implémentation d'une telle interface (*java.io.Closeable*) au sein d'une classe qui ne fait jamais appel à la méthode *close*. Ce qui reflète

un impact sur la mémoire dues aux ressources non-relâchées.

3.2 Métriques logicielles

Dans cette section, nous verrons ce que sont les métriques logicielles, quelles sont leurs intérêts et ensuite nous établirons une liste des métriques pertinentes dans le cadre de ce travail. C'est-à-dire des métriques qui ont un lien avec la *section 3.1 - Code Smells* ainsi qu'aux mesures pouvant avoir un lien direct ou indirect avec l'impact environnemental.

3.2.1 Définition

Une métrique logicielle est une mesure faite sur une propriété d'un logiciel ou de ses spécifications. *Métrique* et *mesure* sont deux termes souvent utilisés comme synonymes bien que les métriques sont les "fonctions" alors que les mesures sont des quantités (nombres) obtenus en appliquant les métriques [14].

Par exemple, une métrique peut être le **nombre de lignes de code** du programme. Si celui-ci contient **10 000** lignes de code, ce nombre correspond à la mesure de la métrique.

3.2.2 L'intérêt des métriques logicielles

Le but des métriques concernant le développement logiciel de manière générale va être de mesurer la qualité du logiciel sur base de différents critères. Ces critères peuvent concerner la sécurité, la portabilité, la complexité, la performance du logiciel, etc. Dans le cadre de ce mémoire, les métriques qui nous intéressent sont celles qui ont un lien avec le code source et c'est la raison pour laquelle la section suivante ne présente que le métriques touchant les critères suivants : *l'efficacité*, *la complexité*, *la compréhension*, *la réutilisation* et *la maintenabilié* du code source du logiciel. En d'autres termes, ces métriques fournissent un moyen d'obtenir une meilleure conception pour du code orienté-objet afin d'avoir un code plus facilement réutilisable et maintenable. Elles servent également à identifier d'éventuelles anomalies et mettent en avant les parties du programme qui nécessitent une modification et une ré-implémentation [17].

Comme le présente la *section 3.1 - Code Smells*, les mauvaises pratiques vont impacter la qualité du code source et nous verrons, dans la section 3.2.3, ces métriques et mesures qui peuvent elles aussi avoir un impact négatif sur la qualité du code.

3.2.3 Métriques concernant la qualité du code source

Complexité Cyclomatique

La complexité cyclomatique (CC) introduite par Thomas McCabe en 1976, est utilisée pour évaluer la complexité d'un algorithme dans une méthode et est définie par le nombre de chemins linéairement indépendants qu'il est possible d'emprunter dans celle-ci [15, 16]. Idéalement, une valeur de complexité cyclomatique doit être inférieure à 10. Au-delà d'une valeur de 20 (CC), du refactoring est recommandé [18].

La taille - Nombre de lignes de code

Le nombre de lignes de code d'une méthode permet d'évaluer la facilité à comprendre le code par les développeurs et les personnes en charge de la maintenance. Cette métrique peut s'appliquer autant pour une méthode, qu'une classe ainsi qu'au projet dans son entièreté. Si l'on considère les méthodes, cette métrique rejoint le code smell "méthodes longues" où l'on considère que la taille d'une bonne méthode doit se situer entre 4 et 20 lignes de code afin d'assurer une compréhension et une maintenance plus aisée.

Pourcentage de commentaires

Une fois que la métrique précédente (taille du code) a été calculée, on peut l'étendre pour inclure le nombre de commentaires. Le pourcentage de commentaires présent est alors calculé en divisant le nombre de commentaires par le nombre de lignes de code totales où l'on a soustrait le nombre de lignes vides [15].

Méthodes pondérées par classe

Les méthodes pondérées (Weighted Methods per Class (WMC)) permettent de mesurer la complexité d'une classe. Il existe deux façons pour la mesurer :

— compter le nombre de méthodes de la classe.
— sommer les complexités cyclomatiques des méthodes.

Plus la valeur WMC obtenue est élevée, plus la classe en question est complexe. Cette valeur peut être utilisée pour prédire le temps et l'effort requis pour développer et maintenir la classe [15]. Pour rappel, plus un code est complexe, plus celui-ci sera difficile à comprendre et à maintenir.

Réponse pour une classe

La métrique **Réponse pour une classe** est le nombre total de méthodes qui peuvent potentiellement être exécutées en réponse à un message reçu par un objet d'une classe. Il s'agit donc du nombre de méthodes de la classe

additionné de toutes les méthodes distinctes qui sont appelées depuis les méthodes de la classe ainsi que les méthodes héritées. Plus ce nombre augmente, plus l'effort requis pour les tests augmente puisque la séquence de tests s'élargit [17].

Manque de cohésion des méthodes

Le manque de cohésion des méthodes (Lack of Cohesion of Methods (LCOM)) mesure le degré de similarité des méthodes grâce aux variables et attributs de la classe. Plus simplement, LCOM permet de détecter si dans une même classe, il n'y a pas plusieurs ensembles de méthodes et variables qui ont des vies entièrement séparées, ce qui permet d'identifier les défauts dans la conception des classes [15]. Voici un exemple parlant d'un manque de cohésion dans les méthodes d'une classe :

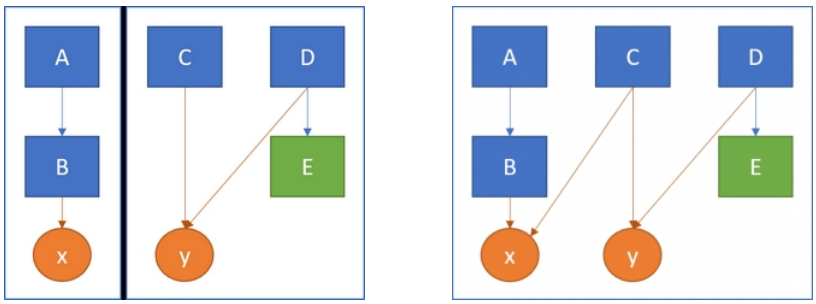


FIGURE 3.7 – Source : Cohesion metrics [19].

Soient A, B, C, D, E des méthodes et x, y des attributs d'une classe. On remarque bien dans l'image de gauche que d'un côté, la méthode A appelle la méthode B qui fait un traitement sur l'attribut x et de l'autre côté, la méthode C et D qui utilisent l'attribut y dans leur traitement et la méthode E qui est appelée par D. On a donc sur la première image de la Figure 3.7 deux ensembles de méthodes et variables qui ont des vies séparées, la valeur du LCOM est donc de 2. La seconde image en revanche, ne forme qu'un ensemble avec une bonne cohésion où LCOM vaut 1.

Une bonne cohésion indique une bonne subdivision de classes et à l'inverse, un manque de cohésion augmente la complexité de la classe [15].

Couplage entre classes d'objets

Le couplage entre classes d'objets (Coupling Between Object Classes (CBO)) correspond au nombre de classes qui sont couplées à une certaine classe. Une classe est couplée à une autre si des méthodes d'une des classes utilisent les méthodes ou attributs de l'autre classe. Une mesure de couplage élevée complique la réutilisation de la classe et indique une complexité plus importante, rendant sa maintenance plus compliqué [15, 17].

Profondeur de l'arbre d'héritage

La métrique de profondeur de l'arbre d'héritage (Depth of Inheritance Tree (DIT)) mesure la longueur du chemin le plus long depuis un noeud jusqu'à la racine de l'arbre. Cette métrique calcule donc à quel point une classe est déclarée dans la hiérarchie d'héritage. Plus une classe est profonde dans la hiérarchie, plus celle-ci hérite d'un nombre de méthodes élevé, ce qui rend le comportement et la conception de la classe plus complexe. Cela induit une moins bonne facilité à comprendre, maintenir et réutiliser la classe [15, 17]. On considère qu'il ne faut pas dépasser une valeur de DIT égale à 5. Sous cette valeur, le principe d'héritage de la programmation est géneralement correctement exploitée [20].

Nombre d'enfants

Cette métrique mesure tout simplement le nombre de sous-classes qui hérite de la classe parent. Cette valeur est un bon indicateur sur la réutilisabilité, basée sur le principe même de l'héritage. D'un autre côté, ce nombre est également une indication possible du niveau de testing nécessaire [21].

3.3 Outils d'analyse statique

L'analyse statique[3] de code permet d'obtenir des informations concernant la qualité du code et du comportement que celui-ci peut avoir lors de son exécution. Cette section présente différents outils permettant la détection des code smells présentés dans ce chapitre. La section 3.3.1 offre une comparaison de tous ces outils détaillant les types de code smell détectés par chaque outil.

JDeodorant est un outil sous forme de Plug-in Eclipse qui permet de détecter automatiquement les code smells *Feature Envy*, *God Class*, *Long Method* et *Switch Statement* dans un environnement Java.

InFusion est un outil sous forme d'application autonome supportant les langages C, C++ et java qui permet d'analyser aussi bien la structure que le code d'un système. L'outil est capable de détecter plus de 20 défauts de conception et code smells.

PMD est un outil pouvant se trouver sous forme de Plug-in Eclipse mais également sous forme d'application autonome qui scanne essentiellement du code source Java permettant de détecter du *code mort*, des *variables locales inutilisées*, de la *duplication de code* ainsi que les code smells *Large Class*, *Long Method* et *Long Parameter List*.

3. Qui ne nécessite pas l'exécution du programme.

iPlasma est une plateforme intégrée permettant d'évaluer la qualité des systèmes orientés-objet dont les langages supportés sont le C++ et le Java. Cet outil permet de détecter les code smells suivant : *Data Class*, *Feature Envy*, *Large Class*, *Shotgun Surgery*, *Refused Bequest*, *Long Method*, *Long Parameter List* et *Speculative Generality*.

Stench Blossom est un détecteur de code smells qui donne un bon aperçu des code smells présents dans le code mais sans offrir un moyen de faire de la rétro-ingénierie de code. Cet outil, qui se trouve sous forme de Plug-in Eclipse permet donc aux développeurs de détecter et de comprendre la source des problèmes dans du code Java. L'outil permet de détecter les code smells *Data Clumps*, *Feature Envy*, *Long Method*, *Large Class*, *Message Chains* et *Switch Statement*. Deux autres code smells *InstanceOf* et *Typecast* peuvent également être détecté par l'outil mais ceux-ci ne sont pas considérés comme tel dans ce travail puisqu'ils n'ont aucun impact sur la maintenabilité, la compréhension du code ou sur l'environnement.

JSpIRIT (Java Smart Identication of Refactoring opportunITies) [25] est un outil flexible qui permet aux développeurs de configurer et d'étendre l'outil en fournissant différentes stratégies pour identifier et classer les code smells. En d'autres termes, des règles et des critères permettent de détecter les code smells avec une certaine priorité en fonction des objectifs des développeurs. Les code smells détectables par JSpIRIT et qui nous intéressent dans ce travail sont *Long Method*, *Long Class*, *Data Class*, *Feature Envy*, *Intensive Coupling*, *Refused Bequest* et *Shotgun Surgery*.

BSDT (Bad Smell Detecting Tool) [26] est un Plug-in Eclipse capable de détecter une série de code smells en utilisant des seuils prédéfinis pour les métriques logicielles. Voici un tableau qui montre les seuils et les métriques utilisés pour chaque code smell que l'outil est capable de détecter [26] :

Code Smell	**Level**	**Metrics & treshold**
Large Class	Class	$NOM > 20 \mid\mid NOF > 9 \mid\mid LOC > 750$
Long Method	Method	MLOC > 50
Long Parameter List	Method	PAR > 5
Switch Statement	Method	VG > 10
Parallel Inheritance	Class	$DIT > 3 \mid\mid NSC > 4$
Data Class	Class	$WMC > 50 \mid\mid LCOM > 0.8$
Lazy Class	Class	$(NOM < 5 \&\& NOF < 5) \mid\mid DIT < 2$

DECOR (DEtection et CORrection) est également un outil permettant la détection et la correction automatique des code smells qui se base sur un ensemble de règles qu'il est nécessaire de renseigner au préalable [27]. L'outil est capable de détecter les code smells *Large Class*, *Lazy Class*, *Long*

Method, *Long Parameter List*, *Refused Bequest* et *Speculative Generality*.

HIST (Historical Information for Smell deTection) [28, 29] est un outil de détection de code smells qui, en plus de l'analyse statique de code, se base sur l'historique d'un gestionnaire de versions. Plus spécifiquement, il analyse les co-changements ayant lieu entre les différents artefacts de code source. L'outil est capable de détecter les 5 code smells suivants : *Divergent Change*, *Shotgun Surgery*, *Parallel Inheritance*, *God Class* et *Feature Envy*.

L'outil a été évalué sur les changements historiques de 8 projets Java tels que Apache Ant, Tomcat, jEdit et 5 autres projets d'API Android. L'étude dédiée à cette évaluation indique une précision allant de 61% à 80% et un rappel se situant entre 61% et 100%. Comparé à des approches alternatives, il a été observé [29] que l'outil HIST tend à fournir de meilleures performances et particulièrement au niveau du recall, qui est capable d'identifier des code smells que les autres approches omettent. Cela est dû au fait que ces autres approches ne considère pas les informations historiques.

CheckStyle est un outil de développement permettant décrire du code Java selon une certaine norme de codage. L'outil est hautement configurable, prenant en charge presque toutes les normes de codage. Ce qui permet de vérifier un grand nombre d'aspects dans le code source. Ce qui permet par exemple d'identifier des problèmes de conception de classe et des problèmes de conception de méthodes [31]. L'outil peut se présenter sous la forme d'un Plug-in Eclipse ou une application autonome et permet la détection des code smells suivants : *Duplicated code*, *Large Class*, *Long Method* et *Long Parameter List* [30].

aDoctor (AnDrOid Code smell detecTOR) est un outil qui a été développé par dessus l'outil de développement Java d'Eclipse (JDT [4]) et permet l'identification de 15 code smells spécifiques à Android. Ces code smells sont présentés dans la section *3.1.7 - Code smells spécifiques à Android*. L'outil a été évalué en utilisant 18 applications [35] et a montré une précision de 98% et un recall de 98% également.

3.3.1 Synthèse des outils d'analyse

Cette section répertorie les différents outils d'analyse sous forme de tableau où chaque outil est associé à un contexte qui représente la forme dans laquelle on peut retrouver l'outil ainsi qu'aux code smells qu'il est capable de détecter.

4. https ://www.eclipse.org/jdt/

Outil	**Contexte**	**Langage**	**Code smells**
JDeodorant	Plug-in Eclipse	Java	Feature Envy, God Class, Long Method, Switch Statement, Duplicated Code
InFusion	Application standalone	Java, C, C++	22 codes smells
PMD	Plug-in Eclipse et Application standalone	Java, JavaScript	Large Class, Long Method, Long Parameter List, Duplicated Code, Dead Code
iPlasma	Application standalone	Java, C++	Data Class, Feature Envy, Large Class, Shotgun Surgery, Refused Bequest, Long Method, Long Parameter List, Speculative Generality
Stench Blossom	Plug-in Eclipse	Java	Data Clumps, Feature Envy, Long Method, Large Class, Message Chains, Switch Statement
JSpIRIT	Plug-in Eclipse	Java	Long Method, Long Class, Data Class, Feature Envy, Intensive Coupling, Refused Bequest, Shotgun Surgery
BSDT	Plug-in Eclipse	Java	Large Class, Long Method, Long Parameter List, Switch Statement, Parallel Inheritance, Data Class, Lazy Class
DECOR	Application standalone	Java	Large Class, Lazy Class, Long Method, Long Parameter List, Refused Bequest, Speculative Generality
HIST	Application standalone	Java	Divergent Change, Shotgun Surgery, Parallel Inheritance, God Class, Feature Envy
CheckStyle	Plug-in Eclipse et application standalone	Java	Duplicated Code, Large Class, Long Method, Long Parameter List
aDoctor	Par dessus l'outil de développement Java d'Eclipse	Android	Les code smells d'Android (cf. 3.1.7)

Chapitre 4

Impact écologique des Code Smells

Dans ce chapitre, nous verrons l'impact écologique que peut avoir un code inefficace ou tout simplement mal conçu. Dans la section 4.1, nous verrons l'impact des code smells de Fowler suivi par la section 4.2 qui présente l'impact des code smells spécifiques aux applications mobiles dont la nature peut être différente des code smells de Fowler. Dans la section 4.3 nous verrons un autre impact qui concerne le réseau pour finir avec la section 4.4 qui synthétise ce chapitre.

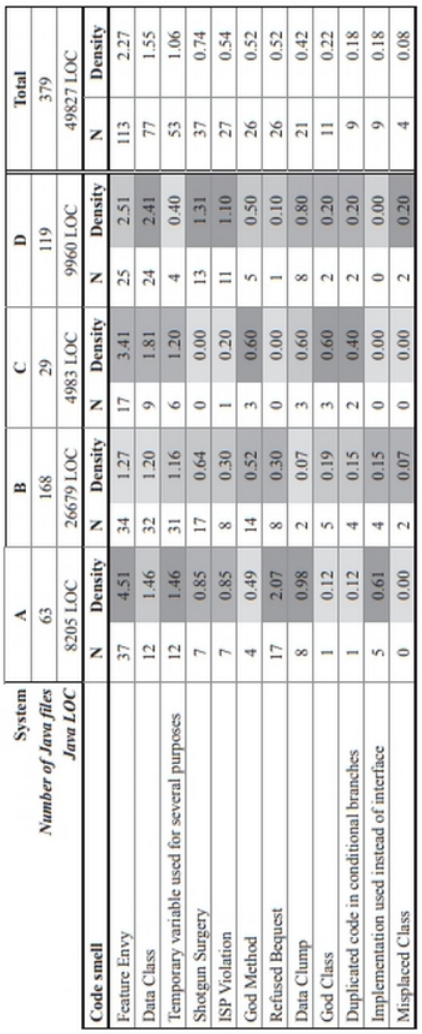
4.1 Impact des code smells de Fowler

Pour rappel, les code smells de Fowler sont des mauvaises pratiques que l'on retrouve essentiellement dans des applications de bureau et auxquelles on reproche principalement le fait de rendre le code plus difficile à comprendre et à maintenir. Bien qu'il n'existe à ce jour aucune preuve que ce type de code smells a un éventuel impact sur la **consommation énergétique** de la machine, ceux-ci ont néanmoins un impact certain sur la qualité du code source. Cet impact, comme annoncé précédemment va rendre le code difficilement maintenable. Lorsqu'un programme doit changer ou tout simplement évoluer, un code difficilement maintenable va rendre l'évolution de ce programme bien plus longue et plus difficile. C'est donc incontestablement du temps de travail supplémentaire qui sera nécessaire pour effectuer ce changement. Étant donné que ce mémoire traite de l'impact écologique que peut avoir le code source d'un programme dans sa globalité, il est tout à fait intéressant de prendre en compte le fait qu'un programme mal conçu, avec la présence de code smells de Fowler par exemple, va finalement demander du temps de travail supplémentaire. Ce travail de maintenance s'effectue sur un ordinateur de bureau ou un ordinateur portable qui consomme inévitablement de l'énergie.

Il est assez difficile d'estimer et d'évaluer l'effort demandé pour maintenir et faire évoluer un code source puisque plusieurs critères rentrent en compte. Par exemple, si la personne en charge de la maintenance n'est pas la même personne qui s'est chargée d'implémenter le système, que le niveau de compétence du développeur n'est pas suffisant ou tout simplement en fonction du nombre de code smells présents dans le code, l'effort à fournir pour maintenir le code peut considérablement être différent en fonction de ces critères.

Une revue systématique [37] de 2011, n'avait déjà trouvé que cinq études traitant de l'impact des code smells sur la maintenance de code. Pour la plupart des études identifiées dans cette revue, celles-ci se concentrent principalement sur des outils et des méthodes utilisés pour détecter automatiquement les code smells. De plus, les résultats de ces études sont peu concluants en vue du manque de preuves sur les mesures qui ont été prises. [36].

C'est la raison pour laquelle une étude [36] a été plus loin dans cette recherche en menant une étude contrôlée pour quantifier la relation entre les code smells et l'effort de maintenance à fournir dans un environnement industriel avec des développeurs professionnels. Bien que l'effort de maintenance en soi n'est pas réellement pertinent dans le cadre de ce travail, il est intéressant de faire le lien entre cet effort et la consommation énergétique que cela engendre. Cette étude s'est donc concentrée sur l'impact que peut avoir 12 code smells (Data Class, Data Clump, Duplication de code, Feature Envy, Large Class, Long Method, Refused Bequest, Shotgun Surgery, Temporary Field, ...) sur l'effort de la maintenance. Pour ce faire, quatre systèmes ont été développés et mis en place. Chaque système dispose d'un nombre de fichiers et d'un nombre de lignes de code différents ainsi qu'une densité de code smells différente, comme le montre la Figure 4.1.



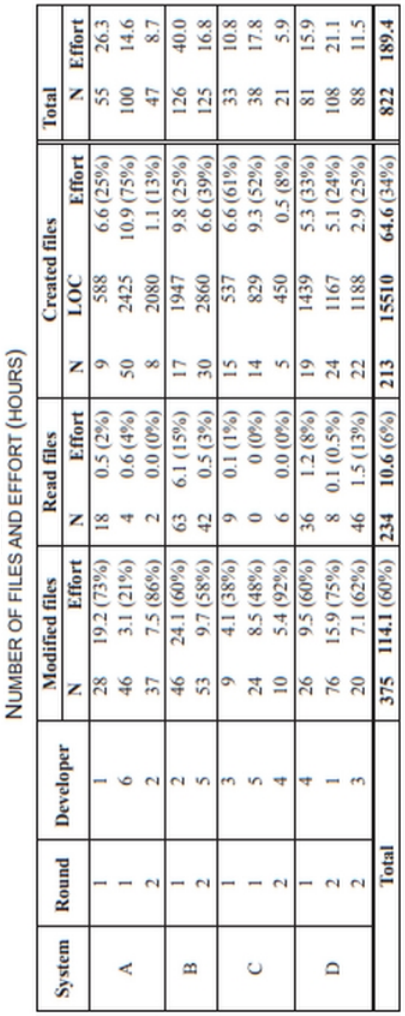
System	A		B		C		D		Total	
Number of Java files	63		168		29		119		379	
Java LOC	8205 LOC		26679 LOC		4983 LOC		9960 LOC		49827 LOC	
Code smell	**N**	**Density**	**N**	**Density**	**N**	**Density**	**N**	**Density**	**N**	**Density**
Feature Envy	37	4.51	34	1.27	17	3.41	25	2.51	113	2.27
Data Class	12	1.46	32	1.20	9	1.81	24	2.41	77	1.55
Temporary variable used for several purposes	12	1.46	31	1.16	6	1.20	4	0.40	53	1.06
Shotgun Surgery	7	0.85	17	0.64	0	0.00	13	1.31	37	0.74
ISP Violation	7	0.85	8	0.30	1	0.20	11	1.10	27	0.54
God Method	4	0.49	14	0.52	3	0.60	5	0.50	26	0.52
Refused Bequest	17	2.07	8	0.30	0	0.00	1	0.10	26	0.52
Data Clump	8	0.98	2	0.07	3	0.60	8	0.80	21	0.42
God Class	1	0.12	5	0.19	3	0.60	2	0.20	11	0.22
Duplicated code in conditional branches	1	0.12	4	0.15	2	0.40	2	0.20	9	0.18
Implementation used instead of interface	5	0.61	4	0.15	0	0.00	0	0.00	9	0.18
Misplaced Class	0	0.00	2	0.07	0	0.00	2	0.20	4	0.08

FIGURE 4.1 – Source : [36].

Les quatre systèmes sont des systèmes Java développés de façon à ce qu'ils soient fonctionnellement équivalents mais indépendants. Ainsi, six développeurs professionnels ont été engagés pour effectuer des tâches de maintenance sur les différents systèmes. Le temps passé sur chacun des fichiers Java était enregistré automatiquement et chaque action effectuée sur les fichiers (modification, création de nouveaux fichiers, ...) était également observée.

Les résultats de cette étude sont intéressants et confirment le fait qu'estimer l'impact des code smells sur la maintenance est compliqué. Par exemple sur la Figure 4.2, ils ont pu remarquer que pour un même système (A) et pour un même Round (1), le développeur 1 a nécessité presque le double

d'effort (26,3 heures) que le développeur 6 (14,6 heures) pour effectuer la même tâche. Ce paramètre suffit à lui seul pour appuyer le fait que le temps nécessaire pour maintenir du code peut différer du simple au double.



NUMBER OF FILES AND EFFORT (HOURS)

System	Round	Developer	Modified files		Read files		Created files			Total	
			N	Effort	N	Effort	N	LOC	Effort	N	Effort
A	1	1	28	19.2 (73%)	18	0.5 (2%)	9	588	6.6 (25%)	55	26.3
	1	6	46	3.1 (21%)	4	0.6 (4%)	50	2425	10.9 (75%)	100	14.6
	2	2	37	7.5 (86%)	2	0.0 (0%)	8	2080	1.1 (13%)	47	8.7
B	1	2	46	24.1 (60%)	63	6.1 (15%)	17	1947	9.8 (25%)	126	40.0
	2	5	53	9.7 (58%)	42	0.5 (3%)	30	2860	6.6 (39%)	125	16.8
C	1	3	9	4.1 (38%)	9	0.1 (1%)	15	537	6.6 (61%)	33	10.8
	1	5	24	8.5 (48%)	0	0 (0%)	14	829	9.3 (52%)	38	17.8
	2	4	10	5.4 (92%)	6	0.0 (0%)	5	450	0.5 (8%)	21	5.9
D	1	4	26	9.5 (60%)	36	1.2 (8%)	19	1439	5.3 (33%)	81	15.9
	2	1	76	15.9 (75%)	8	0.1 (0.5%)	24	1167	5.1 (24%)	108	21.1
	2	3	20	7.1 (62%)	46	1.5 (13%)	22	1188	2.9 (25%)	88	11.5
Total			**375**	**114.1 (60%)**	**234**	**10.6 (6%)**	**213**	**15510**	**64.6 (34%)**	**822**	**189.4**

FIGURE 4.2 – Source : [36].

Ils ont finalement constaté que les fichiers avec la présence des code smells *Feature Envy*, *God Class*, *Temporary Field*, *Shotgun Surgery* et des *implémentations à la place d'interface* étaient associés à un effort plus important que des fichiers ne présentant pas ces code smells. De plus, ils ont constaté que la présence du code smell *Refused Bequest* était associé à une légère réduction sur le temps de maintenance.

De manière théorique, si l'on considère qu'un ordinateur fixe complet consomme en moyenne 200 wattheure (Wh), ce qui représente une consommation énergétique de 200 watts pour une heure d'utilisation, on peut estimer l'impact énergétique d'un travail de maintenance. En se basant sur la Figure 4.2, un total de 189,4 heures a été nécessaire pour effectuer le travail demandé pour l'ensemble des développeurs. À titre d'exemple, cela représente une consommation énergétique de 37 880 W (200 x 189,4), soit 37,88 kW. Étant donné que la production d'électricité [38] dans la filière du nucléaire émet 6 grammes de CO2 pour produire 1 kWh, on peut en conclure qu'un travail de 189,4 heures sur une machine consommant 200 Wh consomme 37,88 kW et émet donc 227,28 g de CO2.

Une autre étude [39] a quant à elle investigué des classes afin de voir si la présence de code smells dans celles-ci fait qu'elles sont plus sujettes aux changements que les classes sans présence de code smells. Les auteurs de cette étude ont, pour ce faire, détecté 29 code smells à l'aide de l'outil *DECOR* dans respectivement 9 et 13 versions des logiciels Azureus [1] (maintenant connu sous le nom de "Vuze") et Eclipse [2]. Ils ont ensuite étudié la relation qu'il pouvait y avoir entre les classes en présence de code smells et les classes sujettes aux changements. Ils montrent ainsi que dans pratiquement toutes les versions d'Azureus et Eclipse, les classes présentant des code smells sont plus sujettes aux changements que les autres classes. Les code smells ont donc bien un impact négatif sur les classes et celui-ci augmente lorsque le nombre de smells augmente au sein de la classe.

Ils ont également constaté que pour chaque logiciel, certains types de code smells spécifiques étaient davantage la cause d'un changement. Pour l'outil d'Azureus, il s'agit notamment des code smells *NotAbstract* (qui est la raison d'un changement dans 7 des 9 versions), *AbstractClass* et *Large Class* (qui sont tous deux responsables d'un changement dans 5 des 9 versions). Pour Eclipse, il s'agit des code smells *HasChildren* [3], *Message Chains* (cf. Section 3.1.5) et *Not Complex* [4]. Cette étude soutient donc finalement que les code smells peuvent avoir un impact négatif sur l'évolution d'un logiciel. Pour rappel, ce type d'impact négatif va influencer le temps et l'effort de maintenance qui, comme nous l'avons vu précédemment demande des ressources supplémentaires. Ces ressources ont indirectement un impact écologique puisque le temps supplémentaire à travailler sur l'évolution du logiciel à cause de la présence des code smells fera consommer davantage d'énergie.

1. `http://azureus.sourceforge.net/`
2. `http://www.eclipse.org/`
3. Décrit des classes avec beaucoup d'enfants
4. Présence de plein de petites classes sans complexité et réalisant chacune une responsabilité atomique

Pour finir cette section sur l'impact écologique des code smells de Fowler, une troisième étude [40] a évalué de manière empirique l'impact énergétique du **refactoring** des code smells. Jusqu'à présent, il était préconisé de corriger les code smells afin de réduire l'**effort de maintenance** et de faciliter l'évolution du logiciel. Dans cette nouvelle et plus récente étude, la question de recherche s'est portée sur l'impact positif ou négatif du refactoring des code smells bien connus que l'on peut retrouver dans les applications standards développées en Java. Selon cette étude [40], plusieurs autres études mettent en avant que le refactoring des code smells a un impact positif sur la maintenabilité du logiciel [43, 44] sans réellement traiter le sujet de l'efficacité énergétique. Et c'est sur ce sujet ainsi que sur la performance des applications, que l'étude en question [40] s'est penchée. Les trois questions de recherches [5] posées sont les suivantes :

— Quel est l'impact du refactoring des code smells sur la consommation d'énergie des applications Open Source Java ?
— Quel est l'impact du refactoring des code smells sur les performances des applications Open Source Java ?"
— Quelles métriques logicielles orientées objet peuvent être de bons indicateurs d'impact sur les performances ou la consommation d'énergie des code smells dans les applications Open Source Java ?

Pour ce faire, les auteurs ont sélectionné cinq différents code smells (Feature Envy, Switch Statement, Long Method, God Class et Duplicated Code) qu'ils ont détecté automatiquement et corrigé (refactoring) dans trois applications Java Open-Source ; Jtrac qui est une application d'environ 14000 lignes de code tandis que les deux autres applications (CashManager et PetClinic) ont chacune environ 2000 lignes de code. Le refactoring s'est fait de deux façons différentes. La première consistant à effectuer le refactoring des code smells de manière individuelle, c'est-à-dire en corrigeant qu'un seul type de code smell sur l'application à la fois et d'analyser son impact. Et la seconde consistant à effectuer le refactoring de tous les code smells en même temps et ce, dans des ordres différents à chaque fois. La Figure 4.3 représente le nombre de code smells corrigés dans les différentes applications et où les abréviations représentent les noms des code smells (FE = Feature Envy, TC = Switch Statement, LM = Long Method, GC = God Class et DC = Duplicated Code). La première ligne du tableau montre le nombre de fois que chaque code smell a été corrigé en ayant appliqué le refactoring du code sur un seul code smell à la fois tandis que les lignes suivantes montrent le nombre de code smells ayant été corrigés en appliquant le refactoring combiné des code smells et où chaque ligne correspond à un ordre différent dans

5. Les questions de recherche ont été traduites en français.

lequel les code smells ont été corrigés. Par exemple, pour le premier tableau, la seconde ligne signifie que le refactoring a eu lieu en corrigeant les code smells dans l'ordre suivant : le code smell Feature Envy a été corrigé (il n'y en avait qu'une occurrence), ensuite le Switch Statement, le Duplicated code et God Class ont été corrigés dans cet ordre avec la même occurrence que pour le refactoring simple. Et pour finir, le code smell Long Method devait être corrigé mais aucune occurrence n'a été corrigé puisque le refactoring des code smells précédents ont permis sa correction.

NUMBER OF CODE SMELL REFACTORINGS FOR THE CASHMANAGER APPLICATION

Combination order	FE	TC	LM	GC	DC
Single Refactoring	1	3	10	2	1
FE-TC-DC-GC-LM	1	3	10	2	0
FE-GC-TC-DC-LM	1	3	0	2	1
FE-GC-LM-DC-TC	1	3	10	2	0
TC-FE-GC-DC-LM	1	3	9	2	1
LM-GC-DC-FE-TC	1	3	10	2	0
LM-DC-GC-FE-TC	1	3	10	2	0
GC-FE-LM-TC-DC	1	3	10	2	0
GC-LM-FE-DC-TC	1	3	10	2	0
GC-DC-FE-LM-TC	1	3	9	2	1
DC-LM-TC-GC-FE	1	3	9	2	1

NUMBER OF CODE SMELL REFACTORINGS FOR THE JTRAC APPLICATION

Combination order	FE	TC	LM	GC	DC
Single Refactoring	8	3	43	11	3
FE-TC-DC-LM-GC	8	3	30	10	3
FE-GC-LM-TC-DC	8	3	31	11	3
FE-GC-DC-TC-LM	8	3	32	11	3
TC-FE-GC-LM-DC	8	3	28	10	3
LM-TC-GC-FE-DC	8	3	43	9	3
LM-TC-GC-DC-FE	8	3	43	9	3
LM-GC-DC-FE-TC	8	3	43	8	3
GC-TC-LM-DC-FE	8	3	30	11	3
DC-FE-LM-GC-TC	8	3	28	9	3

NUMBER OF CODE SMELL REFACTORINGS FOR THE PETCLINIC APPLICATION

Combination order	FE	TC	LM	GC	DC
Single Refactoring	2	-	10	2	-
FE-LM-GC	2	-	10	1	-
FE-GC-LM	2	-	10	2	-
LM-FE-GC	5	-	10	1	-
LM-GC-FE	5	-	10	1	-
GC-FE-LM	5	-	10	2	-
GC-LM-FE	5	-	10	2	-

FIGURE 4.3 – Source : [40].

Les résultats obtenus pour la réponse à la première question ont montré

que pour l'application JTrac, la consommation énergétique est approximativement deux fois moindre lorsque les code smells *Feature Envy* et *Long Method* sont corrigés. (Voir Figure 4.4) où l'abréviation Or correspond à l'application originelle. En revanche, pour les deux autres applications, le refactoring de ces code smells n'apporte pas de gain significatif sur la consommation énergétique.

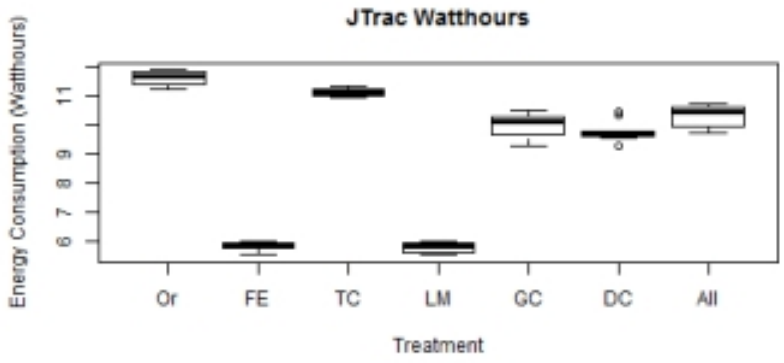


FIGURE 4.4 – Comparaison de la consommation énergétique de l'application avant et après refactoring. Source : [40].

Concernant la performance observée des applications, la seule différence significative a eu lieu avec l'application JTrac où le refactoring individuel des code smells Feature Envy et Long Method a montré une réduction du temps d'exécution de l'application par deux. En revanche, le refactoring des autres code smells ont un léger impact négatif sur la performance de l'application (voir Figure 4.5)

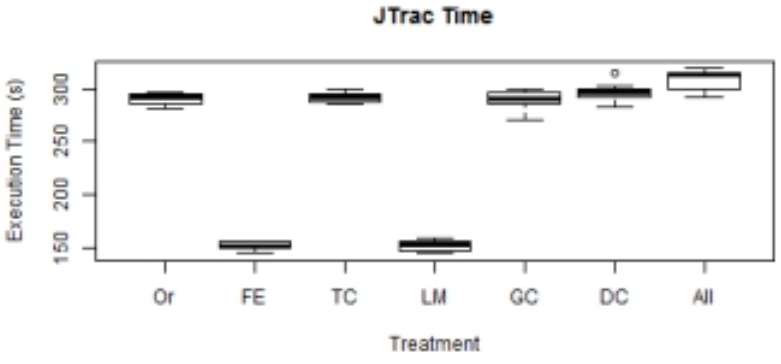


FIGURE 4.5 – Comparaison de la performance de l'application avant et après refactoring. Source : [40].

Finalement, l'étude souligne néanmoins que les mesures sont prises avec des outils logiciels et que des mesures physiques sont nécessaires pour évaluer de façon **fiable** ce genre d'impacts (à l'aide d'un ampèremètre par exemple) car aucun indicateur fiable sur l'impact de la consommation énergétique et la performance du refactoring des code smells ne peut être identifié. Ensuite, ils ont montré que dans les applications d'une ampleur similaire à JTrac (14000 lignes de code), le refactoring des code smells tels que Feature Envy et Long Method a un impact positif et significatif sur la consommation énergétique de l'application. Une autre chose très importante est que l'impact du refactoring des code smells peut être différent selon les code smells. Certains peuvent montrer des améliorations sur les performances, d'autres sur une réduction de la consommation énergétique alors que le refactoring combiné de plusieurs code smells peut empêcher ces améliorations.

Pour conclure cette section, on peut affirmer que la présence de code smells affecte sans aucun doute la maintenabilité du code source comme on vient de le voir à travers ces différentes études. Cet appauvrissement de la maintenabilité mène à un impact environnemental plus important. Et ce, pour les raisons suivantes : principalement parce qu'un code difficilement maintenable implique du travail et du développement plus long, ce qui implique de prendre en compte les activités des développeurs pour mener à bien cette maintenance (leurs déplacements, le chauffage et l'électricité consommés au sein du lieu de travail, tous les appels et vidéos conférences et tout simplement la consommation énergétique des machines des développeurs, ...). Ainsi, au plus le code source est de qualité et bien conçu, au moins la phase de maintenance aura un impact environnemental important. Pour ce faire, les développeurs se doivent de connaître les mauvaises pratiques de programmation et de bien concevoir les logiciels dès le début du processus afin de limiter cet impact.

4.2 Impact des code smells spécifiques à Android

Dans cette section, nous traitons le cadre des applications mobiles sous Android. Ces applications sont exécutées sur des appareils alimentés par des batteries dont la durée de vie fait partie des préoccupations importantes dans ce domaine. C'est la raison pour laquelle il est important d'avoir un code performant et bien conçu. Par exemple, S. Hasan et al. [45] ont étudié l'impact des collections Java et ont découvert qu'utiliser un type de structure de données incorrect peut réduire l'efficacité énergétique de 300%.

Tout comme pour les code smells de Fowler, la connaissance que l'on dispose concernant l'influence des code smells sur la consommation énergétique des applications mobiles est encore assez faible. Une étude très récente, pu-

bliée en janvier 2019 [22] est à la connaissance des auteurs, la plus grosse étude sur le sujet actuellement. Elle cite notamment le travail de A. Carette et al [46] qui ont étudié la relation entre les code smells et l'efficacité énergétique. Mais à nouveau, l'analyse s'est seulement basée sur trois types de code smells en concluant que la correction de ces code smells avait un impact limité sur l'efficacité énergétique où l'on parle d'une amélioration de 4% sur la consommation globale d'énergie. Cette nouvelle étude a donc été menée à grande échelle et s'est portée sur l'influence que peuvent avoir 9 différents code smells spécifiques à Android sur la consommation énergétique de 60 applications Android et ce, de manière bien plus approfondie que les autres études traitant ce sujet. Les 60 applications représentent un total de 19504 méthodes analysées. Le choix des code smells s'est naturellement porté sur ceux ayant théoriquement un lien avec les performances et la consommation énergétique des applications.

Pour mener leurs analyses, les auteurs se sont basés sur deux outils. L'outil aDoctor qui a été présenté à la section 3.3 et qui permet de détecter les code smells spécifiques à Android et le second, PETrA (**P**ower **E**stimation **T**ool fo**r** **A**ndroid) permet d'estimer le profil énergétique des applications mobiles.

Les trois questions de recherches qu'ils se sont posées sont les suivantes :

— Dans quelle mesure les code smells considérés sont diffusés dans les méthodes des applications analysées ?
— Est-ce que les méthodes affectées par les code smells ont une consommation énergétique importante ?
— Est-ce que le refactoring des code smells a un impact positif sur la consommation énergétique des applications mobiles ?

Les résultats obtenus pour la première question de recherche ont montré que l'outil aDoctor a détecté 6155 occurences de code smells parmi les 19504 méthodes. Le code smell *Member Ignoring method* a été détecté 3104 fois, *Slow Loop* a été détecté 1288 fois, *Leaking Thread* a été détecté 828 fois et la transmission de données sans compression a été détecté 564 fois. Les code smells *Inefficient Data Format and Parser* et *Inefficient SQL Query* ont respectivement été détecté 3 et 0 fois. C'est la raison pour laquelle ce dernier code smell a été exclu de l'analyse. Cependant et malgré le peu d'occurrences du code smell *Inefficient Data Format and Parser*, celui-ci a néanmoins été pris en compte dans l'analyse car il peut théoriquement avoir un gros impact sur l'efficacité énergétique. Pour rappel Hasan et al. [45] ont montré dans leur étude qu'un mauvais format de données pouvait réduire l'efficacité énergétique de 300%.

Il y a bien évidemment d'autres code smells Android mais ceux-ci n'ont aucun lien avec la performance ou la consommation énergétique du code source. Comme par exemple les *Public Data* qui posent un problème au niveau de la sécurité mais n'est cependant pas pertinent pour ce travail.

Concernant les résultats obtenus pour la seconde question, qui pour rap-

pel, se demande si les méthodes affectées par les code smells consomment beaucoup dénergie, les auteurs ont d'abord ordonné toutes les méthodes en fonction de ce qu'elles consomment en énergie. Ils ont ensuite observé que parmi les 50% des méthodes (9752) qui consomment le plus, 32% (3120) contenaient un ou plusieurs code smells. Et parmi ces méthodes *malodorantes*,

— 2773 méthodes faisaient parties des 30% les plus consommatrices.

— 1835 méthodes faisaient parties des 10% les plus consommatrices. Ce qui signifie que parmi les 10% des méthodes qui consomment le plus (soit 1950 méthodes), 1835 (soit 94%) avaient une présence de code smells.

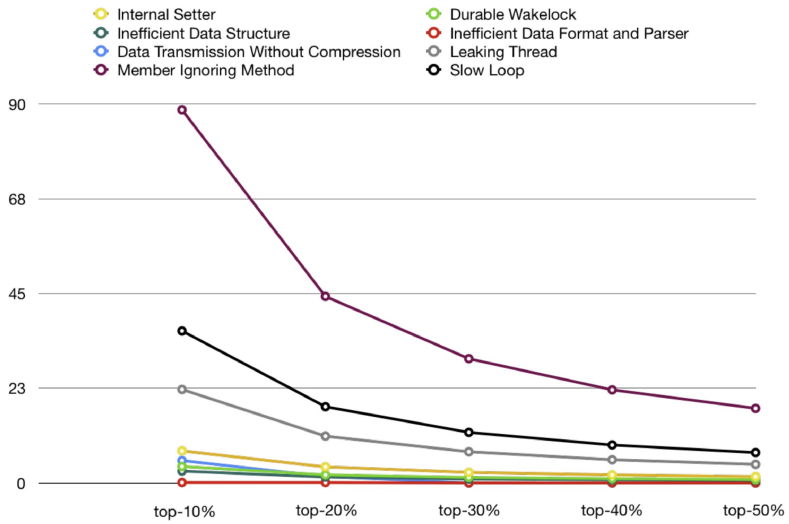


FIGURE 4.6 – Pourcentage des méthodes les plus consommatrices affectées par un ou plusieurs code smells. Source : [22]

Ce résultat laisse supposer qu'une relation existe entre les code smells et la consommation d'énergie. Visuellement, les résultats détaillés de l'analyse sont présentés à la figure 4.6 montrant très clairement que plus l'on se rapproche des méthodes les plus consommatrices en énergie, au plus il y a présence de code smells. Par exemple, la courbe du haut représente la présence du code smell *Member Ignoring Method* et montre qu'en prenant les méthodes dans le top 10 des plus consommatrices en énergie, on trouve près de 90% des occurrences de ce code smell et ainsi de suite pour chaque code smell. Quatre code smells se démarquent particulièrement en étant fréquemment observés dans les méthodes les plus consommatrices. Il s'agit de *Member ignoring method*, *Slow loop*, *Leaking Thread* et *Internal Setter*. D'un autre côté, quatre autre code smells ; *Data Transmission Without Compression*, *Durable Wakelock*, *Inefficient Data Structure* et *Inefficient Data Format and Parser* n'apparaissent que très peu mais la majorité des

occurences de ces code smells sont détectés parmi les 10% des méthodes les plus consommatrices. Ce qui peut être un indicateur concernant un éventuel impact sur la consommation d'énergie.

Cette seconde question de recherche a donc montré que la plupart des code smells analysés avait en quelque sorte une relation avec la consommation énergétique des méthodes affectées par ces code smells. Ce qui nous intéresse maintenant, c'est de pouvoir évaluer cet impact réel que cause les code smells. C'est ce dont traite la troisième et dernière question de recherche. Et pour ce faire, les auteurs ont évalué la mesure dans laquelle le refactoring des code smells avait un effet sur la réduction de la consommation énergétique des méthodes en question. Le refactoring s'est fait manuellement pour 2354 méthodes affectées par un seul code smell considéré, représentant un travail de 450 heures-homme.

Smell Type	Min	1st Qu.	Median	Mean	3rd Qu.	Max
DTWC	0.004	0.006	0.008	0.010	0.013	0.018
R-DTWC	0.004	0.006	0.008	0.010	0.013	0.018
DWL	0.001	0.001	0.001	0.001	0.002	0.003
R-DWL	0.001	0.001	0.001	0.001	0.002	0.003
IDS	0.002	0.002	0.003	0.003	0.003	0.004
R-IDS	0.002	0.002	0.003	0.003	0.003	0.004
IDFAP	0.001	0.001	0.001	0.001	0.001	0.002
R-IDFAP	0.001	0.001	0.001	0.001	0.001	0.002

FIGURE 4.7 – Énergie consommée (en Joule) par les méthodes avant et après le refactoring. Le préfixe "R" signifie que la méthode a été refactorisée. Source : [22]

La Figure 4.7 montre les statistiques résultants concernant la consommation d'énergie des méthodes affectées par les code smells suivant : *Data Transmission Without Compression* (DTWC), *Durable Wakelock* (DWL), *Inefficient Data Structure* (IDS) et *Inefficient Data Format and Parser* (IDFAP) avant et après le refactoring. On peut observer que le refactoring n'a eu aucun effet sur la consommation d'énergie de ces méthodes contenant l'un des code smells précédemment cités, mais les auteurs soulignent néanmoins que leur étude a été menée au niveau des méthodes et que l'impact négatif de certains types de code smells peut très bien se produire à un niveau de granularité supérieur (par exemple, au niveau des classes voire au niveau du projet tout entier). Ensuite, comme annoncé, les 2354 méthodes qui ont été corrigées manuellement comportaient uniquement un code smell

alors que l'étude a montré que les méthodes affectées simultanément par deux code smells apparaissaient plus fréquemment parmi les méthodes les plus consommatrices comme le montre la Figure 4.8. Le refactoring de ces méthodes aurait demandé du travail supplémentaire mais aurait permis de voir si un effet bénéfique peut avoir lieu lorsqu'une méthode est affectée par plus d'un code smell.

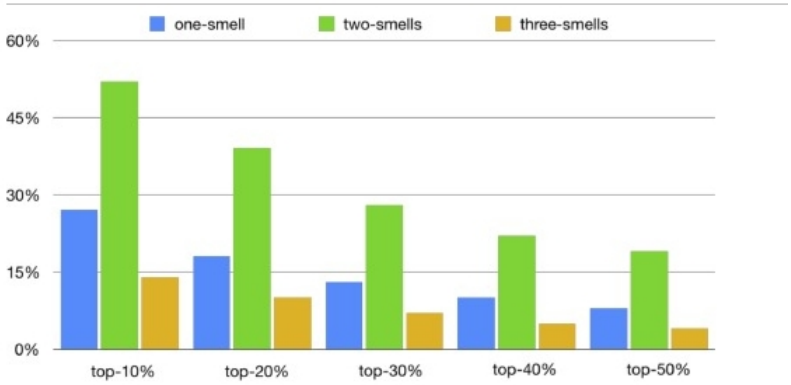


FIGURE 4.8 – Pourcentage des méthodes les plus consommatrices affectées par un ou plusieurs code smells. Source : [22]

L'étude montre donc que le refactoring au niveau des méthodes affectées par un seul code smell ne permet pas d'obtenir une amélioration au niveau de la consommation d'énergie du programme (Ici, il s'agit toujours uniquement des quatre code smells suivants : DTWC, DWL, IDS et IDFAP). En revanche, au niveau de granularité suprieur, tel qu'au niveau des classes et du projet tout entier, le refactoring des code smells permet bel et bien de réduire la consommation énergétique. Tout comme le montre Hasan et al. [45] qui, dans une étude menée dans le cadre d'applications plus vastes et complexes telles que des bibliothèques Java, la présence de structures de données inefficaces comme par exemple un $Hashmap < Integer, Object >$ consomment énormément d'énergie et ce, jusqu'à 3 fois plus que d'autres structures de données plus efficaces.

D'une autre part, les code smells du type *Durable Wakelock* ou *Data Transmission Without Compression* ont un impact évident sur la consommation énergétique mais qui ne se mesurent pas au niveau de la méthode. Par exemple, une méthode acquérant un wakelock (mécanisme indiquant à l'application qu'elle doit rester allumée, ce qui empêche au mobile de "s'endormir" [48]) sans jamais le relâcher, va fatalement avoir un effet négatif sur la consommation énergétique de l'application puisque celle-ci empêchera à l'appareil mobile de se mettre en veille et de passer dans un état permettant d'économiser la consommation et ce, tant que l'application est en cours d'exécution. Et bien entendu, cela ne peut pas se mesurer directement de-

puis la méthode en question, c'est au niveau de l'application que cet impact doit être mesuré. Une explication similaire peut s'appliquer au code smell concernant la compression des données avant de les transmettre, l'impact énergétique de cette mauvaise pratique ne peut pas être mesuré au niveau même de la méthode.

Le même travail a été réalisé concernant les quatre code smells les plus fréquemment observés au sein des méthodes. Il s'agit des code smells *Member Ignoring Method* (MIM), *Slow Loop* (SL), *Leaking Thread* (LT) et *Internal Setter* (IS). Les résultats sont représentés à la Figure 4.9.

Smell Type	Min	1st Qu.	Median	Mean	3rd Qu.	Max
IS	0.076	0.076	0.082	0.083	0.092	0.092
R-IS	0.001	0.001	0.009	0.016	0.024	0.024
LT	0.01	0.02	0.03	0.077	0.013	0.77
R-LT	0.001	0.001	0.003	0.009	0.009	0.019
MIM	0.001	0.002	0.004	0.04	0.028	0.981
R-MIM	0.0001	0.002	0.018	0.02	0.019	0.038
SL	0.001	0.006	0.0119	0.056	0.026	0.929
R-SL	0.001	0.004	0.0114	0.010	0.014	0.018

FIGURE 4.9 – Énergie consommée (en Joule) par les méthodes avant et après le refactoring. Le préfixe "R" signifie que la méthode a été refactorisée. Source : [22]

Les résultats obtenus sont très intéressants puisque l'on peut remarquer une réelle amélioration de l'efficacité énergétique après avoir corrigé les code smells des méthodes. Par exemple, pour le code smell Internal Setter, si l'on considère la moyenne (Mean), on observe une consommation d'énergie de 0.083 Joules. Après le refactoring, on observe une consommation de 0.016 Joules, soit une consommation 5 fois plus faible. L'effet bénéfique du refactoring est encore plus important concernant le code smell Leaking Thread (LT) où l'on peut observer une consommation d'énergie 8 fois inférieure. La refactorisation des deux autres smells permettent tous deux également d'avoir un gain sur la consommation.

Pour conclure cette section, cette étude prouve qu'il y a un réel impact de la présence de code smells dans un code source sur la consommation énergétique des applications Android. Sur les 9 code smells étudiés, un seul n'a jamais été observé et n'a donc pas été considéré (Inefficient SQL Query),

quatre code smells n'ont pas montré d'impact négatif direct sur la consommation au sein uniquement des méthodes mais où d'autres études apportent des résultats concrets sur l'impact de ces code smells à un niveau de granularité supérieur. Et pour finir, les quatre derniers code smells ont quant à eux, un réel impact négatif et direct sur la consommation des méthodes affectées. Le refactoring des code smells permet ainsi d'améliorer l'efficacité énergétique des applications. Les auteurs de cette étude stipulent que dans leurs futures recherches, ils étudieront l'impact des code smells à un niveau de granularité supérieur et souhaitent se concentrer sur un outil de nouvelle génération de refactoring et de verification de la qualité du code.

4.3 Impact environnemental du réseau

Dans les deux sections précédentes, nous avons vu deux types généraux d'impacts : l'impact sur la maintenabilité du programme et l'impact sur la performance du programme (impactant la consommation énergétique de l'appareil). Un troisième impact pertinent dans le cadre de ce travail serait de considérer l'impact qu'un code source peut avoir sur le réseau. C'est-à-dire qu'il est assez fréquent, notamment pour des projets OpenSource [6] de travailler en collaboration sur un même code source. Ce dernier est donc centralisé sur une plateforme telle que GitHub [7] par exemple et le rend accessible par plusieurs personnes. Le problème étant que ce code source va transiter systématiquement entre les machines des développeurs et le serveur d'hébergement de la plateforme en question. D'une part, un code source plus volumineux va solliciter davantage le réseau et ce pour chaque transmission (envoi ou récupération du code source depuis la plateforme). À savoir qu'un réseau informatique est équipé d'équipements informatiques (routeurs, commutateurs, ...) ayant également une certaine consommation énergétique et une empreine carbonne. Donc la présence de code mort ou de la duplication inutile de code par exemple va augmenter la taille du code, ce qui va impacter chaque transmission du code sur ce réseau, augmentant son empreinte carbonne.

Comme nous avons pu le constater dans la section 4.1 sur l'impact des code smells de Fowler, un code source en présence de code smell sera plus sujet au changement et nécessitera davantage de maintenance. Dans le cas où ce code source est partagé entre plusieurs développeurs via une plateforme telle que GitHub, bien plus de transmissions seront effectuées sur le réseau de façon à ce que le code reste à jour pour chaque personne qui travaille sur le code. En d'autres mots, chaque développeur doit récupérer le code depuis le serveur, effectuer les modifications sur ce code et ensuite transmettre à nouveau ce code sur le serveur pour que les autres aient les modifications.

6. Dont le code source est "ouvert" et libre d'accès.
7. https ://github.com/

Donc plus il y a de la maintenance à effectuer sur un code source partagé et plus le nombre de transmissions est multiplié.

En résumé, un code de mauvaise qualité et en présence de code smell peut avoir un impact sur sa maintenabilité, sa performance mais également sur le réseau comme nous venons de le voir dans cette section.

4.4 Synthèse

Dans cette section, nous faisons le point sur l'impact écologique des code smells. La section 4.1, basée sur trois études différentes aura permis d'apprendre plusieurs choses sur l'impact des code smells de *Fowler*.

Premièrement, qu'il n'existe pas de source scientifique avec des résultats fiables sur un éventuel impact énergétique de ceux-ci.

Deuxièmement, nous avons identifié un appauvrissement de la qualité du code en présence de code smells qui impacte l'effort de maintenance. Cet effort est difficilement estimable sur le plan environnemental pour les raisons explicitées dans la section. Pour rappel, la maintenance et l'évolution d'un logiciel nécessite du travail et celui-ci sera généralement plus long et complexe en présence de code smells. Ainsi, les code smells **Feature Envy**, **God Class**, **Temporary Field**, **Shotgun Surgery** et des **implémentations mises en place sans interface** ont été associés à un effort de maintenance plus important. Qui par extension, ont un impact sur l'empreinte écologique dû au travail supplémentaire. En revanche, la présence du code smell **Refused Bequest** a montré une légère réduction sur l'effort de maintenance.

Troisièmement, nous avons pu constater que la présence de code smells au sein des classes avait tendance à rendre ces classes plus sujettes aux changements. Une classe sujette à un changement signifie donc qu'une maintenance va plus rapidement devoir être effectuée rejoignant ainsi le point précédent concernant l'empreinte écologique du code en présence de code smells. Les code smells **NotAbstract**, **AbstractClass**, **Large Class**, **HasChildren**, **Message Chains** et **Not Complex** ont été identifiés comme rendant particulièrement les classes plus sujettes aux changements.

Pour finir avec l'impact des code smells de *Fowler*, nous avons appris que le refactoring des code smells **Feature Envy** et **Long Method** sur une application d'une envergure d'approximativement 14 000 lignes de code, permettait de réduire la consommation énergétique du programme quasiment par deux. Tandis que le refactoring des code smells **Switch Statement**, **God Class** et **Duplicated Code** ont montré une petite diminution de la performance de l'application. Cette diminution de la performance peut donc potentiellement augmenter la consommation énergétique de la machine.

Ensuite, concernant la section 4.2 qui se base sur l'étude la plus grande et la plus récente de l'impact énergétique des code smells spécifiques à An-

droid, nous avons pu constater que les code smells **Member Ignoring Method**, **Slow Loop**, **Leaking Thread** et **Internal Setter** présents dans une méthode ont tous les quatre un impact direct et significatif sur sa consommation énergétique. Contrairement aux code smells **DataTransmission Without Compression**, **Durable Wakelock**, **Inefficient Data Structure** et **Inefficient Data Format and Parser** qui n'ont pas montré d'impact direct sur la méthode et des mesures supplémentaires doivent être réalisées à un niveau de granularité supérieur (Par exemple, au niveau de la classe ou du projet dans son entièreté) afin de voir si oui ou non, ces code smells ont un impact ou pas.

Généralisation des code smells

Depuis le début, les code smells de *Fowler* et les code smells **Android** ont été différenciés tant par leur description que par leur impact. Cependant, certains d'entre eux, sont parfaitement généralisables qu'importe la plateforme. Par exemple, cela n'a pas encore été mentionné jusqu'à présent mais la majorité des code smells de Fowler s'appliquent également dans un environnement Android et ce, pour la raison que ce type de code smells aura tendance à diminuer la qualité d'un code source et à le rendre moins maintenable. Si ceux-ci influent sur la maintenabilité d'un code Java standard, ils influent également sur la maintenabilité d'un code Android. En revanche, tous les code smells spécifiques à Android ne sont pas applicables aux programmes plus standards. Parmi les 15 code smells Android définis, nous pouvons néanmoins en identifier 5 qui sont généralisables aux applications standards. Ces code smells sont **Leaking Thread**, **Slow Loop**, **Data Transmission Without Compression**, **Inefficient Data Structure** et **Inefficient Data Format and Parser**.

— Des fuites au niveau des threads (Leaking Thread) au sein d'une application standard aura bel et bien un impact puisque cela va occuper du CPU. Sachant que plus le CPU sera sollicité, plus la machine devra délivrer de la puissance et aura inévitablement un impact sur la consommation énergétique.
— Des boucles non-optimisées (Slow Loop), que ce soit pour des applications mobiles ou des applications standards, augmenteront le temps d'exécution du programme en le ralentissant.
— L'envoi de données non compressées (Data Transmission Without Compression) sur un réseau aura un impact similaire et ce, qu'importe la plateforme puisque cela ne concerne plus vraiment la consommation énergétique qu'engendre le programme mais plutôt l'impact que cela va produire en sollicitant davantage le réseau.
— Et pour finir, des formats ou des structures de données inefficaces, bien qu'elles puissent être différentes en fonction des langages de pro-

grammation, peuvent également impacter le temps d'exécution ou la mémoire du programme. Cela peut être des types de listes non optimisées, une mauvaise utilisation de format de dates etc.

Les 10 autres code smells définis comme étant spécifiques à Android le sont bel et bien et ne sont pas généralisable aux programmes standards.

Chapitre 5

Refactoring du code source

Ce chapitre se décompose en deux sections ; dans la section 5.1, nous verrons une liste de techniques de refactoring des code smells et dans la section 5.2 nous verrons des outils de refactoring.

5.1 Correction des Code Smells

Bien qu'il existe des outils de refactoring automatiques pour corriger les code smells (cf. Section 5.2), cette section propose les solutions à mettre en place pour éviter la présence des code smells et peut donc servir tant de référence pour l'implémentation du programme que pour sa correction.

Cette section est répartie en différentes catégories représentant des techniques de refactoring différentes pour les différents code smells de Fowler. Des techniques de refactoring [49] sont proposées pour corriger les code smells de Fowler et la dernière catégorie propose des techniques de refactoring pour les code smells spécifiques à Android.

5.1.1 Arranger les méthodes

Les techniques de refactoring présentées dans cette catégorie permettent de rationaliser les méthodes, de supprimer la duplication de code et de préparer les évolutions futures.

Extract Method

La technique d'extraction des méthodes consiste à déplacer des morceaux de code d'une méthode et de les placer dans d'autres méthodes plus approriées. Cela permet d'obtenir un code plus lisible. Il est cependant important de donner un nom de méthode pertinent et qui décrit l'objectif / le rôle de la nouvelle méthode. Ensuite cela permet d'éviter la duplication de code puisque la nouvelle méthode peut être appelée ailleurs dans le code et ne nécessite plus de ré-écrire du code déjà existant.

Code smell corrigé : Comments, duplicated code, Feature Envy, Long Method, Message Chains.

Impact bénéfique : Maintenance

Inline Method

Cette technique de refactoring consiste simplement à retirer la déléguation qu'une méthode peut avoir envers une autre méthode. Lorsque le contenu d'une méthode est plus évident que la méthode en elle-même, l'appel de la méthode peut être remplacée par son contenu. Cela permet ainsi de minimiser le nombre de méthodes qui ne sont pas nécessaires et rend le code plus simple. Un exemple est montré par la Figure 5.1.

```
class PizzaDelivery {
    //...
    int getRating() {
        return moreThanFiveLateDeliveries() ? 2 : 1;
    }

    boolean moreThanFiveLateDeliveries() {
        return numberOfLateDeliveries > 5;
    }
}
class PizzaDelivery {
    // ...
    int getRating() {
        return numberOfLateDeliveries > 5 ? 2 : 1;
    }
}
```

FIGURE 5.1 – Inline Method, Problem - Solution. Source : [49]

Code smell corrigé : Middle Man

Impact bénéfique : En théorie [1], la consommation énergétique puisque plus il y a d'appels de méthodes, plus la complexité du programme augmente et plus cela demande de ressources.

Extract Variable

La technique d'extraction de variables consiste à rendre une expression complexe plus compréhensible. La Figure 5.2 illustre cette technique de refactoring. Il est important de donner des noms de variables corrects qui annoncent clairement ce qu'elles représentent. Cela permet d'avoir un code plus lisible et moins de commentaires interminables.

1. Puisqu'il n'y a à ce jour, aucune véritable preuve scientifique fiable à ce sujet.

```
def renderBanner(self):
    if (self.platform.toUpperCase().indexOf("MAC") > -1) and \
       (self.browser.toUpperCase().indexOf("IE") > -1) and \
       self.wasInitialized() and (self.resize > 0):
        # do something...


Après refactoring...


def renderBanner(self):
    isMacOs = self.plateform.toUpperCase().indexOf("MAC") > -1
    isIE = self.browser.toUpperCase().indexOf("IE") > -1
    wasResized = self.resize > 0

    if isMacOs and isIE and self.wasInitialized() and wasResized:
        # do something
```

FIGURE 5.2 – Extract variable, Problem - Solution. Source : [49]

Cette technique de refactoring introduit plus de variables dans le programme mais cela est contre-balancé par la facilité à lire le code et à réduire la duplication de code.

Code smell corrigé : Duplicated code, comments.

Impact bénéfique : Maintenance

Inline Temp

Inline Temp consiste à remplacer une variable temporaire par son expression directement.

```
def hasDiscount(order):                    def hasDiscount(order):
    basePrice = order.basePrice()   =>         return order.basePrice() > 1000
    return basePrice > 1000
```

FIGURE 5.3 – Inline Temp, Problem - Solution. Source : [49]

Cette technique est pratiquement toujours mis en place lors de la technique de refactoring qui suit (Replace Temp with Query).

Replace Temp with Query

Remplacer une variable temporaire par une méthode qui consiste à placer l'expression mise dans cette variable directement dans une méthode et ensuite de l'appeler. Cela permet de rendre le code plus lisible et compréhensible.

En revanche, cela va impacter la performance du programme puisque des appels des méthodes supplémentaires auront lieu. Bien qu'aucune étude ne prouve concrètement aujourd'hui que cela a un impact notable sur la consommation énergétique, nous savons au moins que la présence de code smell impact la phase de maintenance et d'évolution qui elle va avoir un impact sur le temps de travail nécessaire pour faire évoluer le programme.

Code smell corrigé : Long Method

Impact bénéfique : Maintenance

Impact négatif : Performance et donc en théorie sur la consommation énergétique.

Replace Method with Method Object

Lorsqu'une méthode est trop longue et que l'on ne peut la séparer à cause de variables locales qui sont difficiles à isoler l'une de l'autre, on peut transformer cette méthode en une nouvelle classe séparée. Cela permet de corriger les méthodes trop longues, ce qui rend le code plus facilement maintenable mais l'ajout de classes va impacter la complexité de tout le système, ce qui **théoriquement** devrait réduire les performances et augmenter la consommation énergétique.

Code smell corrigé : Long Method

Impact bénéfique : Maintenance

Impact négatif : Performance et donc en théorie sur la consommation énergétique.

Substitute Algorithm

La substitution d'algorithme au sein d'une méthode consiste simplement à remplacer le corps de cette méthode par l'implémentation d'un nouvel algorithme plus clair et plus performant.

Code smell corrigé : Duplicated Code

Impact positif : Performance et donc impact positif sur la consommation énergétique.

5.1.2 Déplacer des fonctionnalités entre les classes

Les techniques de refactoring de cette catégorie montrent comment déplacer des fonctionnalités entre des classes, créer de nouvelles classes et de masquer des détails d'implémentation des accès publics.

Move Method

Lorsqu'une méthode est plus utilisée dans une autre classe que la sienne, il est bon de créer une nouvelle méthode dans cette autre classe et d'y déplacer le code de l'ancienne méthode.

Code smells corrigés : Alternative Classes with Different Interfaces, Data Class, Feature Envy, Inappropriate Intimacy, Message Chains, Parallel Inheritance Hierarchies et Shotgun Surgery. [13]

Impact bénéfique : Maintenance

Move Field

De manière similaire à la technique précédente, lorsqu'un attribut est plus utilisé dans une autre classe que sa propre classe, il est judicieux de créer cet attribut dans la classe qui l'utilise le plus et de le supprimer de la classe initiale. Évidemment, sans oublier de remplacer les références de l'attribut avec des appels appropriés.

Code smells corrigés : Feature Envy, Inappropriate Intimacy, Parallel Inheritance Hierarchies et Shotgun Surgery. [13]

Impact bénéfique : Maintenance

Extract Class

L'extraction de classes consiste à corriger le fait qu'une classe fait le travail de deux voire de plusieurs classes. Pour ce faire, il suffit de créer une ou plusieurs nouvelles classes et d'y placer les attributs et méthodes dédiés au rôle de la classe. L'introduction du code smell **Obsession des primitives** (cf. 3.1.1) montre un exemple de classe qui contient et manipule des données qui peuvent être propre à plusieurs classes. La Figure 5.4 montre comment cela peut être corrigé.

Cette technique permet de faciliter la maintenance et l'évolution du programme et de respecter le principe d'une seule responsabilité par classe. Attention à ne pas rendre les classes trop petite non plus afin de ne pas se retrouver avec des classes qui n'en font pas assez, ce qui est également une autre mauvaise pratique.

Code smells corrigés : Data Clumps, Divergent Change, Duplicated Code, Inappropriate Intimacy, Large Class, Primitive Obsession et Temporary Field. [13]

Impact bénéfique : Maintenance

Inline Class

À l'inverse de la précédente technique, lorsqu'une classe n'en fait pas assez, qu'elle n'a pas de rôle concret, il est préférable de regrouper les fonctionnalités de cette classe dans une autre classe. Ce qui permet finalement de supprimer des classes dont on a pas forcément besoin, libérant ainsi de la mémoire et réduisant la complexité générale du programme.

```python
class User:

    def __init__(self):
        self.firstname = "Renaud"
        self.lastname = "De Boeck"
        self.phone = Phone("+32", "0472 00 00 00")
        self.address = Address("Rue du Paradis", 16, "1315", "BE")


class Phone:

    def __init__(self, prefix, number):
        self.prefix = prefix
        self.number = number

class Address:

    def __init__(self, street_name, street_num, zip_code, country):
        self.street_name = street_name
        self.street_num = street_num
        self.zip_code = zip_code
        self.country = country
```

FIGURE 5.4 – Refactoring du code smell Obsession de primitives.

Code smells corrigés : Lazy Class, Shotgun Surgery et Speculative Generality [13]

Impact bénéfique : En théorie, la consommation énergétique grâce au gain en complexité et en mémoire.

Hide Delegate

Lorsqu'une classe accède à une classe en passant par une classe intermédiaire pour effectuer une action (Exemple : *MyClass.getA().getB().doSomething()*), la délégation cachée peut être utilisée pour que la classe initiale (MyClass) n'aie plus connaissance ou ne dépende plus de la classe **B**). Pour ce faire, il suffit d'implémenter une méthode dans la classe **A** qui se chargera elle-même de déléguer le travail à la classe B.

Cela permet d'avoir une meilleure indépendance entre les classes. Cependant, ce refactoring peut mener au code smell Middle Man, qui pour rappel, signifique que l'on se retrouve avec une classe ayant trop de méthodes dont le seul rôle est de déléguer...

Code smells corrigés : Inappropriate Intimacy et Message Chains

Remove Middle Man

Pour corriger le code smell Middle Man, qui signifie qu'une classe possède trop de méthodes dont l'objectif est de déléguer, il suffit simplement de supprimer ces méthodes et de faire en sorte que la classe appelante, appelle directement la méthode finale (ayant un rôle autre que la déléguation).

Cela permet d'éliminer le code smell **Middle Man** mais mème à avoir la présence d'autres smells tels que **Inappropriate Intimacy** et **Message Chains**.

Code smell corrigé : Middle Man

Introduce Foreign Method & Local Extension

Ces deux techniques de refactoring consistent simplement à ajouter une méthode (Foreign Method) ou de créer une nouvelle classe (Local Extension) dans le système afin de résoudre le problème de la classe de librairie incomplète. Ces deux techniques ne sont pas pertinente dans le cadre de ce travail puisque celles-ci ne font que rajouter de la complexité au programme et ne contribuent donc en rien à une amélioration de la consommation énergétique du programme.

Code smell corrigé : Incomplete Library Class

5.1.3 Organiser les données

Les techniques de refactoring de cette catégorie permet de mieux gérer les données du programmes et ainsi de rendre les classes plus facilement réutilisables et portables.

Replace Data Value with Object

Cette technique de refactoring est un cas particulier de l'*Extract Class* qui dans ce cas consistait à séparer les multiples responsibilités présentes. Dans le cas du remplacement des valeurs de données par un objet, il s'agit de remplacer un attribut de type primitif (int, string, ...) devenu trop complexe de part son utilisation par une nouvelle classe. De plus, ce genre d'attributs (voir Figure 5.5) peut facilement se retrouver dans plusieurs classes, ce qui engendre de la duplication de code.

Dans le cas où le type de la variable est un tableau contenant une série de données et que l'utilisation de ce tableau sert de stockage de données ou que son utilisation est fréquent et plus complexe qu'un simple tableau, l'utilisation d'une classe où les éléments du tableau deviennent des attributs permet une meilleure compréhension et utilisation du code.

Code smells corrigés : Large Class, Primitive Obsession [13] et possiblement le smell Duplicated Code [49].

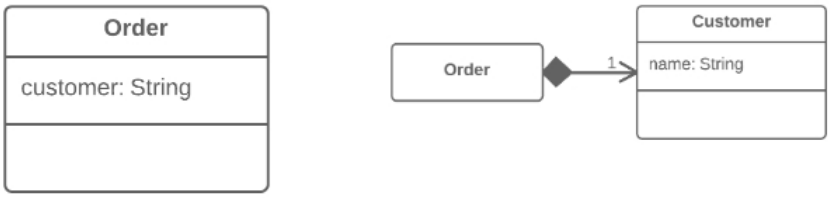


FIGURE 5.5 – Refactoring du code smell Replace Data Value with Object [49]

Impact bénéfique : Maintenance

Change Bidirectional Association to Unidirectinal

Lorsque deux classes ont une relation bidirectionnelle entre-elles mais qu'une des deux classes n'utilise pas les fonctionnalités de l'autre classe, alors il est préférable de retirer cette relation bidirectionnelle et ne laisser qu'une relation unidirectionnelle. Cela permet de simplifier la classe qui n'a pas besoin de cette relation, ce qui permet d'avoir moins de code et ainsi moins de code à maintenir.

Code smell corrigé : Inappropriate Intimacy
Impact bénéfique : Maintenance

Encapsulate Field & Collection

L'encapsulation des attributs consiste à rendre les attributs publics en privés et de créer des méthodes (Getter et Setter) permettant d'avoir un contrôle sur la récupération et le changement des valeurs de ces attributs. Concernant les **collections**, telles que les listes, il n'est pas recommandé de créer de simples assesseurs (Get et Set) mais plutôt de créer une méthode permettant d'ajouter un élément dans la collection, une autre permettant d'en retirer un et pour finir, une méthode qui retourne une copie de la collection de façon à ce qu'elle ne soit pas modifiable. Cette technique d'encapsulation permet d'avoir les attributs et les méthodes comportementales au sein de la même classe, ce qui facilite notamment la maintenance du code mais il n'est pas sans savoir que l'ajout des méthodes augmente légèrement la complexité globale du programme pouvant théoriquement augmenter sa consommation énergétique dû à une augmentation d'appels de méthode. Cependant, l'importance est d'avoir un code de qualité, respectant les principes de la programmation orientée-objet permettant une meilleure maintenance et évolution du programme sur le long terme. D'autant plus qu'aucune étude à ce jour n'apporte de preuves concrètes sur la consommation énergétique que peuvent engendrer de tels code smells. De

ce point de vue, il est intéressant d'en convenir que quelques appels de méthodes supplémentaires dans un programme sont négligeables par rapport aux heures de maintenance et d'évolution supplémentaires qui peuvent être nécessaires lorsque le programme n'est pas bien conçu.

Code smell corrigé : Data Class
Impact bénéfique : Maintenance
Impact négatif : En théorie, la consommation énergétique

Replace Type Code with Class

Lorsqu'une classe contient un ensemble d'attributs de type primimtif utilisé sous forme de *Type Code* (ce qui signifique que ces attributs représentent finalement un même concept mais contiennent des valeurs différentes - Voir Figure 5.6), il est recommandé de créer une classe pour représenter ce concept et de créer une relation entre la nouvelle classe et l'ancienne.

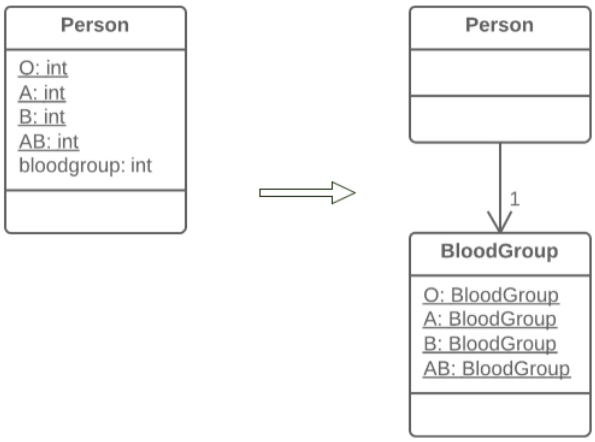


FIGURE 5.6 – Refactoring du code smell Replace Type Code with Class [49]

Cela permet un meilleur contrôle des valeurs que peuvent avoir, dans ce cas-ci (Figure 5.6), le groupe sanguin d'une personne. Autrement, en manipulant des entiers pour représenter le groupe sanguin, des erreurs peuvent rapidement se présenter si l'on se trompe de valeur pour un groupe en particulier. Toute valeur entière est théoriquement acceptée alors qu'avec une nouvelle classe, seul une valeur de groupe sanguin est acceptée.

Code smell corrigé : Primitive Obsession
Impact bénéfique : Maintenance

Replace Type Code with Subclasses

De la même manière que la technique précédente, lorsque des attributs sont utilisés pour caractériser davantage la classe, il est préférable de créer des sous-classes à cette dernière (Figure 5.7). Ce qui améliore le principe de responsabilité unique des classes et rend le programme plus lisible. En revanche, l'ajout de classes augmente la complexité du programme mais à l'avantage que si l'on désire évoluer le programme en ajoutant un nouveau type, il suffit simplement de créer une nouvelle classe sans toucher au reste du code. De plus, le polymorphisme peut être utilisé, évitant le code smell *Switch Statement*.

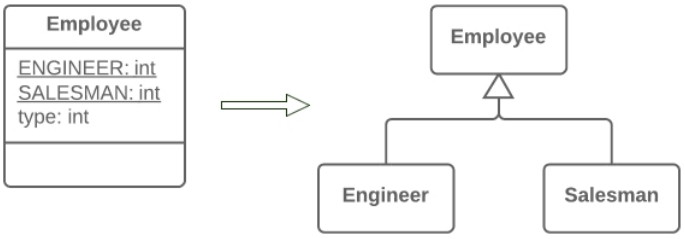


FIGURE 5.7 – Refactoring du code smell Replace Type Code with Subclasses [49]

Code smells corrigés : Primitive Obsession et Switch Statement
Impact bénéfique : Maintenance

Replace Type Code with State/Strategy

S'il n'est pas possible d'utiliser l'héritage comme dans la technique de refactoring précédente, une solution est de remplacer les attributs (Type Code), par un objet *state* (Figure 5.8). Cela est notamment utile lorsque la valeur du *Type Code* change au cours du temps (Une personne est stagiaire puis devient vendeuse etc.). De même, si une nouvelle valeur doit être ajoutée, il suffit d'ajouter une nouvelle sous-classe à notre état sans modifier le code déjà existant, ce qui est une bonne chose lors de l'évolution d'un programme.

Code smells corrigés : Primitive Obsession et Switch Statement
Impact bénéfique : Maintenance

5.1.4 Simplifier les expressions conditionnelles

Les techniques de refactoring de cette catégorie proposent des solutions pour rendre les expressions conditionnelles moins complexes et donc plus lisibles.

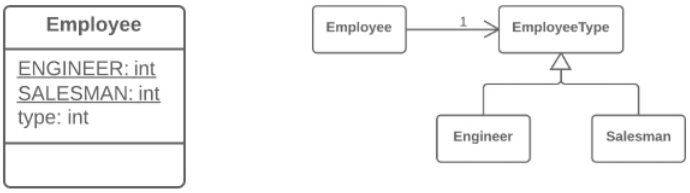


FIGURE 5.8 – Refactoring du code smell Replace Type Code with State/Strategy [49]

Decompose Conditional

La décomposition conditionnelle permet de décomposer une condition complexe à l'aide de méthodes. Plus un code est long et complexe, plus il sera difficile à comprendre et donc plus compliqué à faire évoluer. En plaçant les instructions et les expressions dans des méthodes avec un nom explicite, cela offre une meilleure compréhension du code et permet, sur le long terme, de revenir sur celui-ci sans devoir le décortiquer pour comprendre ce qu'il fait réellement.

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

Après le refactoring :

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

FIGURE 5.9 – Technique de refactoring Decompose Conditional [49]

Comme le montre la Figure 5.9, avant le refactoring, il faut comprendre l'expression conditionnelle et puis il faut comprende ce que font chacune des instructions exécutée dans les deux cas de la condition. En remplaçant ces instructions par des méthodes, il est plus facile de comprendre que si nous sommes en été, il suffit d'appliquer les charges applicables en été, sinon il suffit d'appliquer les charges d'hiver. L'idée est de simplifier au maximum la méthode et le code en général, ce qui permet une maintenance de code bien plus facile. Cependant, l'ajout d'appels de méthodes va augmenter la complexité du programme.

Code smell corrigé : Long Method
Impact bénéfique : Maintenance
Impact négatif : En théorie, sur la consommation énergétique dû à l'augmentation d'appels de méthodes.

Replace Conditional with Polymorphism

Dans le cas d'une méthode qui possède une série de if-elseif-else ou de switch case exécutée sur un type d'objet qui se prête à l'orienté-objet, il est recommandé de créer des sous-classes et d'utiliser le polymorphisme à la place. L'avantage à cela est que lorsque l'on souhaite ajouter une nouvelle variante d'exécution, il suffit d'ajouter une nouvelle sous-classe avec la méthode que chacune des classes exécute grâce au polymorphisme et donc le code existant ne doit pas être modifié. Cela évite de devoir trouver chaque conditions équivalentes dans le code pour ajouter la nouvelle variante.

cette technique de refactoring permet une maintenance plus aisée mais augmente la complexité générale du programme suite à l'augmentation du nombre de classes.

Code smell corrigé : Switch Statement
Impact bénéfique : Maintenance

5.1.5 Simplifier les appels de méthodes

Les techniques de refactoring de cette catégorie proposent des solutions permettant de rendre les appels de méthodes plus simples et plus faciles à comprendre.

Rename Method

Une technique très simple, il s'agit de renommer les méthodes n'ayant pas de nom explicite. Un nom de méthode qui explique ce que fait celle-ci, permet de rendre le code plus lisible.

Code smells corrigés : Comments, Speculative Generality et Alternative Classes with Different Interfaces
Impact bénéfique : Maintenance

Remove Parameter

La technique de supprimer des paramètres consiste simplement à retirer tous les paramètres que peuvent prendre les méthodes et qui ne sont pas utilisés dans le corps de la méthode. Même si cela est pour anticiper un futur besoin, ce n'est pas nécessaire de laisser des paramètres inutiles. Cela

rajoute inutilement de le complexité dans le programme et complexifie la maintenance du code.

Code smell corrigé : Speculative Generality, Dead Code
Impact bénéfique : Maintenance

Replace Parameter with Explicit Methods

Lorsque le contenu d'une méthode est partitionné en fonction de ses paramètres, ce qui s'apparente à un Switch Statement, il est préférable d'extraire chaque partie de la méthode et d'en faire une nouvelle méthode que l'on appellera. (Voir Figure 5.10)

FIGURE 5.10 – Technique de refactoring Replace Parameter with Explicit Methods [49]

Cela permet d'améliorer la lisibilité du code.

Code smell corrigé : Switch Statement
Impact bénéfique : Maintenance

Preserve Whole Object

La préservation de l'objet en entier consiste à passer un objet en paramètre à une méthode et non à y passer ses attributs. Par exemple, supposons que l'on ait un objet *Square* qui a deux attributs *height* et *width*. Au lieu de récupérer les valeurs des attributs et puis de les passer à une méthode pour effectuer une opération, il suffit d'y passer l'objet tout entier. Par exemple, si la méthode en question nécessite plus de données à un moment, il ne faudra pas chercher et modifier tous les emplacements dans le code où l'on passe ces paramètres à la méthode, tout se passera toujours à un seul endroit, la méthode elle-même. Ce qui facilite grandement la maintenance du programme.

Code smells corrigés : Data Clumps, Long Method et Long Parameter List

Impact bénéfique : Maintenance

Introduce Parameter Object

L'introduction de paramètres sous forme d'objet consiste à remplacer des groupes d'attributs similaires par une classe. Par exemple, l'utilisation des variables **Date de début**, **Date de fin** et **Durée** peuvent être utilisé au sein d'une classe **DateRange** qui se chargera de représenter ces variables. Cela permet d'avoir un code plus lisible et éviter la manipulation d'un groupe de données. La compréhension du code est par ce fait améliorée.

Code smells corrigés : Data Clumps, Long Method, Long Parameter List et Primitive Obsession
Impact bénéfique : Maintenance

5.1.6 Utiliser l'héritage

Les techniques de refactoring de cette catégorie proposent des solutions permettant d'utiliser au mieux le principe d'héritage afin d'éviter certaines mauvaises pratiques.

Pull Up Field & Method

Lorsque plusieurs classes possèdent le même attribut ou la même méthode, il suffit simplement de le ou la remonter dans une classe mère (à créer si celle-ci n'exsite pas). Ce qui permet de nettoyer la duplication de code.

Code smell corrigé : Duplicated Code

Push Down Field & Method

À l'inverse du Pull Up, le Push Down consiste à redescendre un attribut ou une méthode qui n'est utilisée que par une seule sous-classe. Cela permet d'améliorer la cohérence des classes et de retrouver les attributs et les méthodes là où elles sont supposées être.

Code smell corrigé : Refused Bequest

Collapse Hierarchy

La technique de réduction de la hiérarchie consiste à fusionner une classe et sa sous-classe lorsque celles-ci sont pratiquement identiques. Cela permet de réduire la complexité du programme.

Code smells corrigés : Speculative Generality et Lazy Class

Replace Delegation with Inheritance

Lorsqu'une classe contient de nombreuses méthodes simples qui délèguent à toutes les méthodes d'une autre classe, la solution est de rentre la classe qui délègue, une sous-classe de l'autre classe rendant la déléguation inutile. La Figure 5.11 illustre clairement la situation.

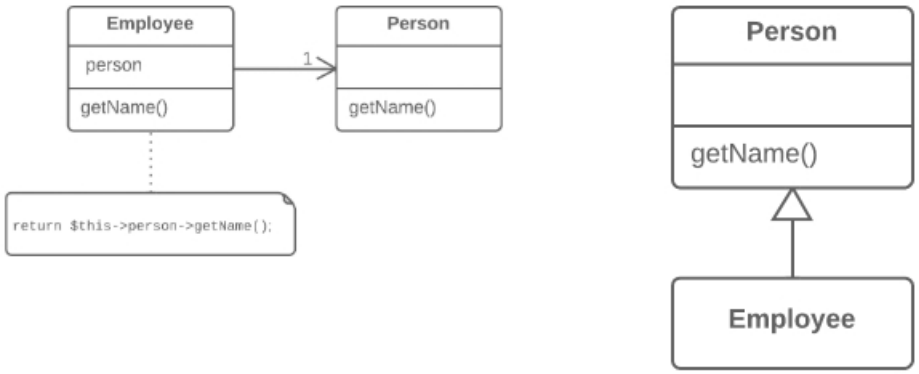


FIGURE 5.11 – Technique de refactoring Replace Delegation with Inheritance [49]

Code smell corrigé : Middle Man

5.2 Outils de refactoring

Parmi les outils de détection de code smells présentés à la section 3.3, l'outil **JDeodorant** permet également le refactoring automatique des code smells *Feature Envy, God Class, Long Method et Switch Statement.* Pour rappel, cet outil est disponible sous forme de Plug-in Eclipse.

JRefactory [51] est un outil gratuit disponible via plusieurs environnement de développement tels que jEdit, Netbeans, JBuilder X, Ant et est également disponible sous forme d'application stand-alone. L'outil permet d'effectuer les techniques de refactoring suivantes : *Move Class, Rename Class, Add Abstract parent class, Add child class, Remove empty class, Extract interface, Push up field, Push down field, Rename field, Push up method, Push up abstract method, Push down method, Move method, Extract method et Rename Parameter.*

FaultBuster [52] est un ensemble d'outils capable de supporter le refactoring automatique de code smells. L'outil est disponible sous forme de plug-in pour les IDEs Eclipse, Netbeans et IntelliJ IDEA. L'outil dispose d'une quarantaine d'algorithmes de refactoring différents [52], il a été testé sur des systèmes de plus de 5 millions de lignes de code au total et a permis de fixer plus de 11 000 problèmes de codage offrant ainsi des solutions complexes et complètes pour améliorer considérablement leur qualité.

IntelliJ IDEA [53] est un environnement de développement java et commercial qui supporte les techniques de refactoring suivantes : *Rename and Move Class/Method/Field*, *Change method signature*, *Extract Method*, *Inline Method*, *Introduce variable/field*, *Inline local variable*, *Extract interface*, *Extract superclass*, *Encapsulate fields*, *Pull up members*, *Push down members* et *Replace inheritance with delegation.*

RefactorIt est un logiciel commercial qui supporte le refactoring automatique des techniques suivantes : *Rename and Move Class/Method, Encapsulate field, Create Factory Method, Extract Method, Extact superclass/interface, Pull up/Push down members.*

Eclipse [54] est un environnement de développement facilement paramétrable et offrant un large ensemble de possibilités de refactoring. L'IDE permet par exemple d'extraire automatiquement des variables locales, des méthodes, des classes ou de déplacer des éléments. Sans oublier que de nombreux plug-in sont disponibles et offrent également cette possibilité d'effectuer du refactoring de code smells (cf. JDeodorant).

Chapitre 6

Synthèse et modèle des impacts écologiques des code smells et de leur refactoring

Ce chapitre a pour but de synthétiser les trois précédents chapitres en faisant, pour chaque code smell, le lien entre le code smell lui-même, son impact et les techniques de refactoring associées (cf. section 6.1). La section 6.2 propose un modèle visuel représentant les influences des différents types de code smells sur l'environnement qui se base sur la section 6.1. Nous verrons quel est l'objectif et l'utilité d'un tel modèle et comment celui-ci a été construit et peut être utilisé.

6.1 Synthèse

Cette section va donc reprendre chacun des code smells que nous avons défini et nous ferons le lien avec le problème qu'il engendre et la ou les techniques de refactoring associées. Les problèmes engendrés par les code smells seront soit un problème de maintenance, soit un problème de performance, soit un problème qui impacte le réseau ou pour finir, un problème impactant directement la consommation énergétique de l'appareil comme nous avons pu le voir avec les code smells Android. Sachant que comme nous avons pu le constater dans le chapitre 4, la maintenance, un problème impactant les performances ou le réseau impacte indirectement la consommation énergétique.

Long Methods

Les méthodes longues sont des méthodes disposant de bien trop de lignes de code et de complexité qu'elles ne devraient.

Problème : Cela **impacte** la *maintenabilité* de ces méthodes dû à une lisibilité moindre et un temps de compréhension plus important. Plus la méthode sera longue et complexe, plus la maintenance le sera.

Solution : Les techniques de refactoring associées à ce code smell sont **Extract Method** qui consiste simplement à déplacer le code de la méthode dans d'autres méthodes ou dans de nouvelles méthodes plus appropriées. Ce qui permet de rendre le code plus lisible et d'augmenter la **maintenabilité** de la méthode. **Replace Temp with query** qui consiste à remplacer une variable temporaire par une nouvelle méthode qui renvoie l'expression contenue dans cette variable. Cela permet de rendre le code plus lisible et compréhensif mais a le désavantage de réduire la performance du programme causé par un nombre d'appels de méthodes plus important. **Replace Method with Method Object** qui consiste à remplacer une méthode trop longue et indissociable par une classe, ce qui offre une meilleure maintenabilité mais augmente aussi la complexité du programme, ce qui diminue sa performance. **Decompose Conditional** qui consiste à décomposer une condition complexe à l'aide de méthodes ayant des noms parlant permettant de rendre le code plus facilement compréhensif, améliorant la maintenabilité mais dont l'augmentation d'appels de méthodes aura un impact sur la performance du programme. **Preserve Whole Object** qui consiste simplement à passer un objet en paramètre d'une méthode et non pas ses attributs. Ce qui permet une meilleure maintenabilité pour l'évolution du programme. Et **Introduce Parameter Object** qui consiste à remplacer un groupe d'attributs liés par une nouvelle classe. Permettant une meilleure manipulation des données et une meilleure compréhension du code. Favorisant ainsi une meilleure maintenabilité.

Donc, les *méthodes longues* ont un **impact négatif** sur la maintenabilité du code mais un **impact positif** sur sa performance. Le refactoring de ce code smell, aura donc un impact positif sur la maintenabilité mais un impact négatif sur la performance et donc sur la consommation énergétique dans certains cas.

Large Class

Les classes larges sont des classes contenant beaucoup trop d'attributs, de méthodes et de lignes de code en général.

Problème : Ces classes disposent bien souvent trop de responsabilités **impactant** la *maintenabilité* des classes de la même manière que les méthodes longues. Et donc plus les classes sont longues, complexes et disposent de responsabilités, au plus l'impact sera important.

Solution : Les techniques de refactoring associées à ce code smell sont

Extract class et **Replace Data Value with Object** qui consistent à remplacer des attributs et des méthodes par la création d'une nouvelle classe qui est plus pertinente et permet de conserver une responsabilité unique pour chaque classe. Il faut que le création d'une nouvelle classe soit justifiée et que celle-ci ne devienne pas le code smell *Lazy Class* ou *Data Class* par exemple.

Primitive Obsession

L'obsession des primitives est l'utilisation excessive des variables de type primitive à travers l'ensemble du programme pour gérer et représenter un ensemble des données.

Problème : Cette pratique **impacte** la manipulation des données et la compréhension générale du programme.

Solution : Les techniques de refactoring associées à ce code smell sont **Extract Class** et **Replace Data Value with Object** que nous l'avons vu pour le code smell Large Class qui permet de corriger l'obsession des primitives au sein d'une classe en y créant une nouvelle classe. **Replace Type Code with Class/Subclass/State** qui est sont des cas particuliers du code smell Replace Data Value with Object finalement et qui sont tous les trois des façons différentes de conception en créant de nouvelles classes. **Introduce Parameter Object** consistant à remplacer les groupes d'attributs par une classe.

Long Parameter List

Les longues listes de paramètres sont des méthodes prenant un nombre important de paramètres.

Problème : Cela rend moins lisible, moins compréhensible et complique l'utilisation et la refactorisation de la méthode. **Impactant** donc la maintenabilité de celle-ci.

Solution : Les techniques de refactoring associées à ce code smell sont **Preserve Whole Object** qui consiste à passer un objet en paramètre à la place de ses attributs et **Introduce Parameter Object** qui consiste à créer une classe qui contiendrait comme attributs, les différents paramètres de la méthode. Pour peu que ce soit cohérent bien entendu. Une fois la classe créée, il suffit d'appliquer la technique de refactoring Preserve Whole Object.

Data Clumps

Les agrégats de données sont des groupes de variables primitives manipulés ensemble à la place de manipuler un seul objet.

Problème : Cela **impacte** la maintenabilité du code et complique son évolution.

Solution : Les techniques de refactoring associées à ce code smell sont **Preserve Whole Object** et **Introduce Parameter Object** exactement de la même façon que le code smell Long Parameter List, sauf que dans ce cas-ci, ce ne sont pas des paramètres à une méthode mais directement des groupements de variables au sein du code.

Switch Statement

L'opérateur Switch ou la série de if-then-else est un code smell qui, comme son nom l'indique, représente un Switch Case ou une longue série de if-else if et dont l'utilisation de cet opérateur n'est pas idéal à la situation.

Problème : Ce code smell viole le principe du Ouvert / Fermé, peut engendrer de la duplication de code et rend le code plus difficile à maintenir.

Solution : Les techniques de refactoring associées à ce code smell sont **Replace Type Code with Subclasses**, **Replace Type Code with State/Strategy**, **Replace Conditional with Polymorphism** et **Replace Parameter with Explicit Methods** qui sont toutes des techniques qui consistent à gérer les différents cas que peut avoir un Switch case à l'aide du principe d'héritage et de polymorphisme.

Temporary Field

Des champs temporaires sont les attributs d'une classe qui ne sont utilisés que dans une voire dans quelques méthodes de la classe. Les valeurs de ces attributs sont pour la plupart du temps vides ou non pertinentes. Ces attributs devraient uniquement se retrouver sous forme de variable au sein de la méthode qui les utilise.

Problème : Cela **impacte** la maintenabilité du code dû notamment à la compréhension plus compliquée du code.

Solution : La technique de refactoring associée à ce code smell est **Extract Class** dans le cas où l'attribut temporaire n'a finalement pas sa place au sein de la classe. Dans le cas contraire, il est nécessaire de remplacer

cet attribut par une variable se retrouvant uniquement dans la méthode qui l'utilise.

Refused Bequest

Le leg refusé est une classe qui hérite d'une autre classe sans utiliser voire très peu les membres de celle-ci.

Problème : Cela peut introduire de la confusion lors de l'évolution du code et rend le code peu élégant, ce qui **impacte** la maintenabilité du code.

Solution : La technique de refactoring associée à ce code smell est **Push Down Field & Method** qui consiste à *descendre* un membre de la classe mère dans la ou les quelques sous-classes étant les seules à l'utiliser.

Alternative Classes with different Interfaces

Les classes alternatives avec différentes interfaces sont l'utilisation de deux voire même de plusieurs classes ou fonctions ayant des noms différents mais qui sont similaires et disposent d'une même responsabilité.

Problème : Cela a pour **impact** d'introduire de la redondance de code et augmente donc l'effort de maintenance.

Solution : Les techniques de refactoring associées à ce code smell sont **Move Method** et **Rename Method**. L'idée étant de déplacer la ou les méthodes similaires au sein d'une et une seule méthode. De même pour les classes similaires où le but est de ne garder qu'une seule classe et de retirer la redondance afin de faciliter à nouveau la maintenance.

Divergent Change

Un changement divergent est un code smell qui représente un changement qui nécessite plusieurs changements dans plusieurs méthodes d'une même classe.

Problème : Il s'agit d'un problème de conception **impactant** la maintenabilité du code.

Solution : La technique de refactoring associée à ce code smell est **Extract Class**

Shotgun Surgery

Le Shotgun Surgery est quant à lui un code smell qui représente un changement qui nécessite plusieurs changements mais cette fois-ci dans plusieurs classes simultanément.

Problème : Cela **impacte** la maintenabilité du code

Solution : Les techniques de refactoring associées à ce code smell sont **Move Method**, **Move Field** et **Inline Class** qui sont trois techniques consistant à déplacer certains membres (méthodes, attributs) dans une classe plus pertinente pour eux.

Parallel Inheritance Hierarchies

Les hiérarchies d'héritage parallèle est un code smell qui implique qu'à chaque fois qu'une sous-classe est ajoutée à une classe, une autre sous-classe doit systématiquement être ajoutée à une autre classe. Introduisant de la duplication de code et une organisation de plus en plus lourde.

Problème : La présence de ce code smell va **impacter** la maintenabilité du code causé par la présence d'un nombre bien plus important de classes.

Solution : Les techniques de refactoring associées à ce code smell sont **Move Method** et **Move Field**.

Comments

Concernant le code smell des commentaires que nous avons dfini, celui-ci n'a pas plus d'impact en tant que tel. Comme expliqué, Il peut camoufler du mauvais code mais en tant que tel, celui-ci n'a pas d'impact sur la maintenance ou la consommation énergétique du code.

Solution : Les techniques de refactoring associées à ce code smell sont **Extract Method** et **Extract Variable** qui permettent toutes les deux de refactoriser le code de façon à le rendre plus compréhensible afin de rendre les commentaires obsolètes et **Rename Method** qui permet de de rendre le nom d'une méthode suffisamment clair pour comprendre ce qu'elle fait.

Duplicated Code

La duplication de code est un code smell relativement fréquent et comme son nom l'indique, représente du code qui a été copié-collé à différents endroits du programme. La duplication de code peut également être considérée

lorsque des morceaux de code sont très similaires.

Problème : Cela **impacte** la *maintenabilité* du code et alourdit inutilement la taille du code source ce qui aura un impact supplémentaire sur le *réseau* en cas de transmission du code sur une plateforme, ...

Solution : Les techniques de refactoring associées à ce code smell sont **Extract Class**, **Extract Method**, **Extract Variable**, **Substitute Algorithm** et **Pull Up Field & Method**

Lazy Class

Une classe paresseuse est une classe ayant une trop petite responsabilité pour ce qu'elle coûte. Ce genre de classe peut apparaître lorsque le code smell agrégats de données est corrigé dans le but d'améliorer la maintenabilité mais l'apparition de cette classe va inutilement augmenter la complexité du programme.

Problème : Le problème n'est ici pas la maintenabilité mais la complexité du programme. Dans des programmes conséquents avec de nombreuses lignes de code ($> 10\ 000$), cette complexité va significativement diminuer les performances du logiciel, **impactant** sa consommation énergétique.

Solution : Les techniques de refactoring associées à ce code smell sont **Inline Class** qui permet de supprimer les classes inutiles en déplaçant leur contenu dans d'autres classes plus appropriées et **Collapse Hierarchy** qui permet de fusionner des classes presque identiques.

Data Class

Les classes de données sont des conteneurs de données qui ne contiennent donc que des attributs et des méthodes ne servant qu'à accéder à ces attributs.

Problème : L'impact est similaire au code smell Lazy Class, bien qu'ici, les opérations sur les données de la classe ne se font pas directement au sein de la classe, ce qui engendre une maintenance plus difficile.

Solution : Les techniques de refactoring associées à ce code smell sont **Move Method** qui consiste à placer la méthode opérante sur les données à l'endroit où se trouvent les données et **Encapsulate Field & Collection** qui consiste à mettre les attributs en privés et de créer les méthodes qui manipuleront ces attributs.

Dead Code

Un code mort est la présence d'attributs, de classes, de fonction ou tout simplement des bouts de code qui ne sont tout simplement jamais utilisés au sein du programme.

Problème : Cela ne fait que perturber la *maintenance* du code. De plus, ce surplus de code augmente la taille du projet, ce qui peut alourdir la transmission de données sur le *réseau* lorsque le code du projet doit par exemple être téléchargé sur des plateformes, servers,...

Solution : Les techniques de refactoring associées à ce code smell sont **Inline Class** ou **Collapse Hierarchy** dans le cas où nous sommes confrontés à une classe inutilisée et **Remove Parameter** pour supprimer les paramètres non nécessaires par exemple. Pour le reste, il suffit simplement de supprimer le code et les fichiers qui ne sont pas utilisés.

Speculative Generality

La généralité spéculative est parfaitement similaire au Code mort, il s'agit de code non utilisé mais volontairement implémenté afin de répondre à un éventuel besoin futur.

Problème : L'impact est identique au code smell *Dead Code*.

Solution : Les techniques de refactoring sont donc identiques au code smell Dead Code.

Feature Envy

Le code smell Feature Envy représente une méthode accédant plus aux membres d'une autre classe que de sa propre classe, ce qui signifie peut-être que cette méthode n'est pas au bon endroit.

Problème : Ce code smell **impacte** la *maintenabilité* du code.

Solution : Les techniques de refactoring associées à ce code smell sont **Extract Method**, **Move Method** et **Move Field** qui, comment nous pouvons le constater, sont trois techniques qui consistent à déplacer du code à un endroit plus approprié et permettant dès lors d'empêcher qu'une méthode accède plus souvent aux membres des autres classes plutôt que de sa propre classe.

Inappropriate Intimacy

L'intimité inappropriée correspond à des classes qui se connaissent beaucoup trop entre-elles.

Problème : Les classes ont un couplage trop important, ce qui complique leur utilisation et leur réutilisation, ce qui **impacte** notamment l'*évolution* du programme.

Solution : Les techniques de refactoring associées à ce code smell sont **Move Method**, **Move Field**, **Extract Class** qui permettent de déplacer des attributs et des méthodes dans les classes appropriées, **Hide Delegate**, **Remove Middle Man** et **Change Bidirectional Association to Unidirectinal**

Message Chains

Les chaînes de messages sont des code smells qui représente des appels de méthodes chaînés parmi plusieurs classes, ce qui les rend dépendantes l'une des autres.

Problème : Le fait que des classes soient dépendantes va **impacter** l'évolutivité du programme ainsi que la réutilisation de ces classes.

Solution : Les techniques de refactoring associées à ce code smell sont **Extract Method**, **Move Method**, **Hide Delegate** et **Remove Middle Man** sont toutes les quatre des techniques offrant la possibilité de supprimer l'enchaînement d'appels qui forme cette chaîne de messages. Où il s'agit de déplacer du code ou de faire en sorte que la méthode appelante appelle directement la dernière méthode de la chaîne.

Middle Man

L'homme du milieu est un code smell qui représente une classe qui ne fait que déléguer une action à une autre classe.

Problème : Ce type de classe n'a aucune autre responsabilité et rajoute tant de la *complexité* au sein du programme que de la *maintenance*. Puisque l'on se retrouve avec des appels de méthodes supplémentaires.

Solution : Les techniques de refactoring associées à ce code smell sont **Remove Middle Man** et **Replace Delegation with Inheritance** qui consistent à supprimer les méthodes qui délèguent le message au sein de la chaîne.

Internal Setter

Dans un environnement Android, le code smell Internal Setter est l'utilisation des setters au sein même de la classe. Cette opération est assez coûteuse et a un **impact direct** sur la consommation énergétique de l'application.

Solution : La seule *technique* ici est simplement de remplacer l'appel de méthode par une assignation directe.

Leaking Thread

Qu'importe l'environnement dans lequel on se trouve, des Threads qui ne s'arrêteront pas avant la fin de l'exécution du programme vont **impacter** la mémoire mais peuvent très bien solliciter le CPU également.

Solution : La technique est donc de s'assurer systématiquement que chaque Thread lancé s'arrêtera au bout d'un moment.

Member ignoring Method

Une méthode ignorant les membres de sa classe est un code smell représantant, comme son nom l'indique, une méthode qui n'utilise jamais les attributs et les autres méthodes de la classe. tant donné que le fait de rendre cette classe *statique* augmente son efficacité, on peut dire que la présence de ce code smell impacte l'efficacité de la classe.

Solution : La technique de refactoring est de rendre ce type de méthodes statiques.

Slow Loop

Les boucles lentes sont des opérations permettant de faire des boucles dans un programme mais qui ne sont pas aussi efficaces que d'autres opérateurs. De manière plus générale, des boucles non-optimisées vont **impacter** significativement les performances du programme et donc sa consommation énergétique.

Solution :Les techniques sont dans un premier temps d'utiliser les opérateurs les plus efficaces, selon les langages. Comme par exemple l'utilisation d'un foreach à la place d'un simple for lorsqu'il est possible est bien plus efficace. D'autre part, le contenu de la boucle se doit également d'être optimisé afin de ne pas boucler davantage que ce qui est nécessaire.

Data Transmission without Compression

La transmission de données sans compression est comme son nom l'indique, l'envoi de donn'ees sur un réseau sans que celles-ci soit compressées. Cela **impacte** fatalement le réseau en le sollicitant davantage.

Solution :La technique de refactoring est de s'assurer que les données soient compressées avant une transmission.

Durable Wakelock

Le wakelock d'une application mobile est important et permet notamment de mettre l'appareil en "veille" de façon à limiter sa consommation énergétique. Le code smell Wakelock Durable signifie que le verrou du wakelock n'est jamais relâché à la fin de l'exécution des tâches du programme, ce qui va donc impacter la consommation énergétique de la batterie.

Solution : La technique est donc de s'assurer que chaque méthode prenant le contrôle du verrou le relâche bien à la fin de son exécution.

Inefficient Data Structure / Format and Parser

Dans la même idée que pour les boucles lentes, l'utilisation de formats ou structures de données inefficaces va **impacter** la performance du programme.

Solution : La technique est donc d'utiliser les formats et structures de données pertinentes à chaque situation et d'utiliser bien entendu les plus optimisés.

Leaking Inner Class

Ce code smell ne concerne que les applications Android et représente une classe interne à une autre. Cette classe interne est utilisée sous forme de tâche asynchrone à la classe externe et le problème est que cette opération peut créer des fuites de mémoire dans le cas o l'activité de la classe externe est détruite alors qu'une tâche asynchrone est toujours en cours d'exécution. Cela va donc **impacter** la mémoire vive et peut **impacter** les performances également dans le cas où la tâche asynchrone exécute continuellement des instructions.

Solution : La technique est dès lors de s'assurer qu'une tâche asynchrone soit arrêtée lorsque l'activité en question est détruite.

No Low Memory Resolver

Ce code smell concerne également les applications Android et signifie que le comportement d'une application s'exécutant en arrière plan ne dispose pas de la méthode *Activity.onLowMemory* permettant de nettoyer les caches et les ressources inutiles de l'application. Ce qui peut donc **impacter** la mémoire. Sachant que l'alourdissement de l'application peut engendrer un impact sur la consommation énergétique de celui-ci puisqu'elle sera en demande de plus de ressources.

Solution : La technique est donc de s'assurer que l'activité principale de l'application mobile dispose de la méthode citée plus haut de façon à optimiser l'utilisation du cache et des ressources.

Rigid Alarm Manager

À nouveau typique à Android, le code smell du Gestionnaire d'alarme rigide signifie qu'une application contient une instance de la classe AlarmManager avec un réglable précis sur le *setRepeating()* ce qui est très drainant pour la batterie. Il est donc recommandé d'utiliser cette méthode lorsqu'une précision sur le temps du rappel est réellement nécessaire.

Solution : La technique de refactoring associée à ce code smell est simplement l'utilisation d'une autre méthode moins précise à cet effet telle que *setInexactRepeating()* qui permet à Android de synchroniser plusieurs alarmes répétable de façon inexacte et les appelle toutes au même moment. L'utilisation de cette méthode, comme on vient de le citer, doit être utiliser uniquement lorsque cela est nécessaire.

6.2 Modèle des impacts écologiques des code smells et de leur refactoring

Cette section présente le modèle qui a été réalisé sur base de tout ce qui a été vu dans les chapitres précédents et montre l'influence du code source sur l'environnement en fonction de différents critères (Code smells, impacts, ...). Ce modèle s'apparente donc à un diagramme d'influence.

6.2.1 Objectif et utilité du modèle

Le modèle, qui est présenté à la section 6.2.2 a plusieurs objectifs. En premier lieu, il permet de montrer l'influence que peut avoir le refactoring sur les code smells et l'influence que peuvent avoir la complexité et la taille du code sur les divers impacts que nous avons identifiés dans le chapitre 4. Et deuxièment, ce modèle a pour utilité de servir comme base théorique pour de

futurs travaux concernant le développement d'un outil logiciel permettant d'identifier automatiquement l'impact environnemental d'un code source et de proposer des solutions pour corriger ou réduire cet impact.

Dans le cadre de ce mémoire, le modèle se veut volontairement abstrait, simple et intuitif. Celui-ci peut finalement servir de prémisse pour de futurs travaux et se doit d'être suffisamment simple et abstrait pour que la connaissance de l'état de l'art de ce mémoire soit bien comprise et interprétée. Il peut également servir à d'autres chercheurs pour entreprendre et élaborer des recherches plus avancées sur ce sujet.

6.2.2 Le modèle d'influence

Définition : un modèle d'influence ou diagramme d'influence est une représentation graphique d'un problème de décision. Il décrit les éléments clés telle que la situation actuelle, les décisions qui peuvent être prises, les incertitudes et les objectifs représentés par des noeufs de formes et de couleurs différentes. Des flèches représentent les influences entre ces noeufs. [56].

Pour des raisons de lisibilité et afin de permettre d'apporter une précision supplémentaire, deux modèles sont proposés. Le second modèle ne fait que reprendre une partie du premier modèle en y ajoutant une précision. Dans le premier modèle (Figure - 6.1), on retrouve quatre types de noeufs différents ; le noeuf bleu foncé qui représente la situation actuelle, le noeuf vert qui représente une décision à prendre, les noeufs bleus clairs qui correspondent aux résultats de la décision (que l'on appelle aussi des variables chances ou incertitudes) et les noeufs rouges correspondent aux différents objectifs.

Explication du modèle : la situation actuelle est le fait d'avoir un code de mauvaise qualité avec présence de code smells. Ce noeud est lié à une décision qui dans notre cas correspond au fait d'effectuer le refactoring des code smells. Cette décision aura une influence sur la présence ou non des différents code smells. Et pour finir, la présence ou non des code smells aura une influence sur les objectifs que l'on vise, c'est-à-dire un plus petit impact sur la maintenabilité, la performance, le réseau et plus globablement, sur l'impact environnemental du logiciel. Le modèle est donc très simple à comprendre, on part du postula que nous avons un code de mauvaise qualité, si l'on décide d'effectuer un refactoring sur ce code, cela va permettre de jouer sur la présence des code smells dans ce code et ainsi bénéficier d'un impact moindre. Dans ce modèle, tous les code smells sont reliés à tous les impacts potentiels et c'est la raison pour laquelle un deuxième modèle a été produit pour ajouter une précision sur l'impact réel des code smells. On peut remarquer également sur les deux modèles que les impacts de la maintenabilité, la performance et le réseau, ont un lien vers l'impact environnemental puisque comme nous l'avons vu à travers ce mémoire, une faible maintenabilité, une faible performance du logiciel et une sollicitation accrue du réseau vont tous

les trois avoir un impact environnemental.

Pour ce faire, le modèle à la Figure 6.2 regroupe différemment les code smells de façon à visualiser leur(s) impact(s) respectif(s). En dehors des code smells, la taille et la complexité d'un code source ont été ajouté au même titre que les code smells.

On voit dès lors que la taille d'un code source a une influence sur la maintenabilité du code et sur le réseau (plus la taille sera importante, moins le code sera maintenable et plus le réseau sera sollicité en cas de transmissions). Les code smells de Fowler ont tous un impact sur la maintenabilité du code. Les code smells Duplicated Code, Dead Code, Speculative Generality, Data Transmission Without Compression ont tous un impact sur le réseau (en plus de la maintenabilité puisque ces code smells sont des code smells de Fowler). Les code smells Lazy Class, Slow Loop, Data Class, Message Chains, Middle Man, Alternative Classes with Different Interfaces et Inefficient Data Structure / Format ont quant à eux un impact sur la performance du code (en plus de la maintenabilité), les code smells Android ont un impact sur la consommation énergétique de l'appareil et pour finir, la complexité du code joue un rôle sur la performance et la consommation énergétique du logiciel. Ce qui résume finalement, de manière visuelle, la connaissance tirée des études que nous avons vu au chapitre 4.

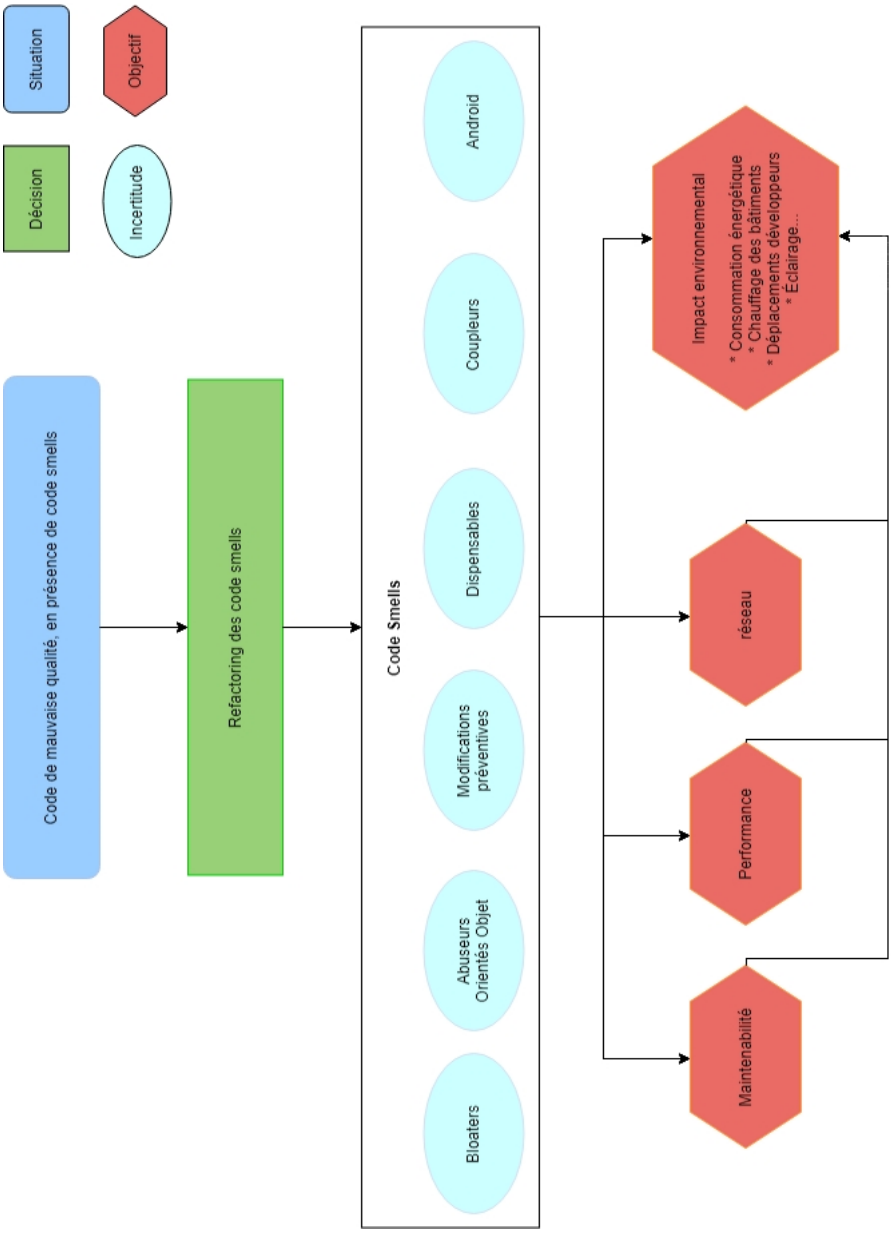


FIGURE 6.1 – Modèle d'influence générique

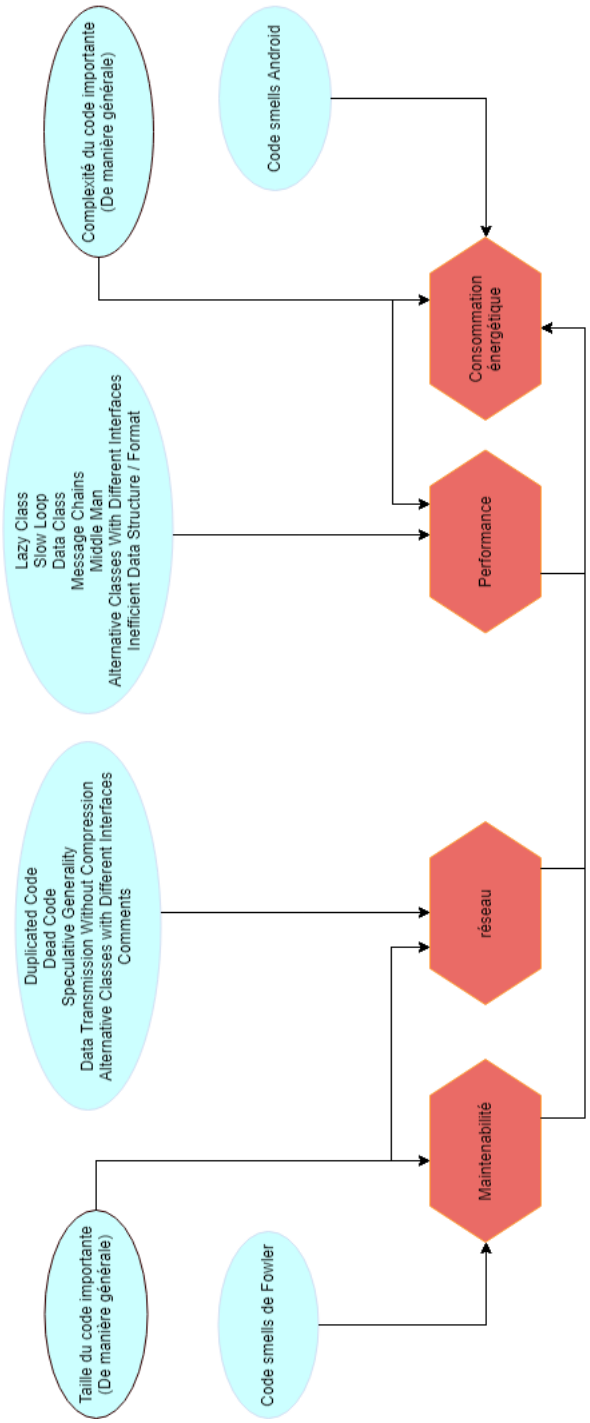


FIGURE 6.2 – Modèle d'influence

Chapitre 7

Conclusion

L'objectif premier de ce mémoire consistait à identifier l'impact environnemental que peuvent avoir les mauvaises pratiques logicielles sur base des sources existantes dans la littérature. Nous avons ainsi pu identifier une série de mauvaises pratiques que l'on nommera Code Smells et dont plusieurs impacts peuvent y être associés. Malgré le peu d'études sur le sujet, il a été montré que les code smells ont bel et bien un impact. Une première étude montre bien que la présence de certains code smells de Fowler dans un code source augmentait l'effort de maintenance de ce dernier. D'autres code smells plus spécifiques comme les boucles lentes ou une mauvaise utilisation des structures de données réduisent quant à eux la performance du programme. Une seconde étude a montré que plus un code source contenait de code smells de Fowler et plus celui-ci était sujet au changement. La troisième étude que nous avons traitée montre à son tour que pour les code source volumineux, le refactoring des code smells de Fowler apporte un gain significatif sur le temps d'exécution et donc sur la consommation énergétique du programme. Nous avons également été plus loin en montrant qu'un impact sur la maintenabilité et la performance du code source a aussi un impact sur l'environnement. Concernant les code smells spécifiques à Android, une seule et grande étude a été analysée et montre que la présence de code smells a un impact significatif sur la consommation énergétique de l'appareil.

Le second objectif du mémoire était de synthétiser la connaissance obtenue de l'état de l'art et d'y apporter un modèle permettant de visualiser plus clairement les enjeux des code smells au sein d'un code source. Ce qui permet de voir plus clairement, chaque impact sur chaque famille de code smells.

7.1 Point de vue critique

Dans cette section, un point de vue critique est posé sur ce mémoire afin de voir quelles sont ses forces et ses faibles et comment pourrait-il être amélioré.

Objectivement, la force de ce mémoire réside dans le fait qu'il est relativement simple à comprendre, et ce, même pour des personnes qui ne maîtrisent pas le domaine du logiciel. Il est assez aisé de comprendre à travers ce mémoire qu'il existe des mauvaises pratiques logicielles pouvant être appliquées par les développeurs, ayant des répercutions sur l'utilisation ou la maintenabilité et l'évolution du code. De cette façon, le mémoire peut autant servir à des développeurs de toutes catégories (débutants comme experts) afin d'optimiser la qualité de leur code source, tout comme il peut servir à des chercheurs pour poursuivre des recherches dans ce domaine.

Au contraire, sa simplicité et l'abstraction utilisée pour le modèle font sa faiblesse. Dans le sens où le modèle peut servir de première base pour l'implémentation d'un outil capable de détecter automatiquement la présence de code smells (tout comme d'autres outils sont déjà capables de le faire) mais sachant apporter une estimation de l'impact que causent les code smells détectés. Au stade actuel, le modèle ne dispose pas assez de précision pour pouvoir directement se lancer dans un tel projet et mériterait d'être amélioré (cf. Section 7.2).

7.2 Perspectives futures

Pour aller plus loin dans la démarche concernant l'impact écologique des code smells, il serait particulièrement intéressant de mettre en place un moyen permettant de mesurer, à l'aide d'outils physiques, la consommation énergétique d'un code source. Plus concrètement, de futurs travaux pourraient mettre en place un ou plusieurs systèmes plus ou moins conséquents en terme de taille (comme a pu le faire, l'une des études que nous avons analysé) et d'effectuer des mesures physiques de manières à analyser la consommation énergétique d'une méthode particulière, d'une classe particulière ainsi que du projet dans son entièreté et ce, avant et après la phase de refactoring. Ce travail permettrait d'apporter des mesures fiables permettant d'identifier de manière plus précise, l'impact énergétique direct que pourraient avoir les code smells et plus particulièrement, les code smells de Fowler spécifiques aux applications de bureau.

Ensuite, le modèle proposé au chapitre 6 pourrait servir de base théorique à la conception d'un outil capable d'estimer ou simplement d'informer les développeurs des erreurs de conceptions et de développement introduites dans le code source. Informant ainsi les développeurs des problèmes dans le code source pouvant impacter le programme d'un point de vue écologique.

Cependant, comme mentionné dans la section 7.1, ce modèle n'est à ce jour pas suffisamment précis et nécessiterait des améliorations. Notamment, il serait intéressant d'incorporer les différentes techniques de refactoring dans le modèle et d'y faire apparaitre les code smells individuellement. Cette amélioration, offrirait une aisance pour l'implémentation d'un futur logiciel afin d'intégrer dans celui-ci, les code smells l'un après l'autre en fonction de leur impact, de leur type et des techniques de refactoring qui lui sont associées.

Bibliographie

[1] Green peace : Impact environnemental du numérique
https://www.greenpeace.fr/il-est-temps-de-renouveler-internet/

[2] The Shift Project. Pour une sobriété numérique : Le nouveau rapport du Shift sur l'impact environnemental du numérique.
https://theshiftproject.org/article/pour-une-sobriete-numerique-rapport-shift/

[3] Frédéric Bordage. Green IT
https://www.greenit.fr/

[4] Analyse statique de programmes.
https://fr.wikipedia.org/wiki/Analyse_statique_de_programmes

[5] Master MVA de lENS Paris-Saclay. Data Analytics Post (DAP) L'impact énergétique du numérique.
https://dataanalyticspost.com/limpact-energetique-du-numerique/

[6] Wikipedia - IBM
https://fr.wikipedia.org/wiki/IBM

[7] Wikipedia - Code Smells
https://fr.wikipedia.org/wiki/Code_smell

[8] Alexander Shvets. Gerhard Frey. Marina Pavlova - Code Smells
https://sourcemaking.com/refactoring/smells

[9] Mohamed Aladdin. Write clean code and get rid of code smells.
https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318

[10] Mntyl, M. V. and Lassenius, C. "Subjective Evaluation of Software Evolvability Using Code Smells : An Empirical Study". Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431.

[11] M. Fowler, "Refactoring : Improving the Design of Existing Code", Canada : Addison-Wesley, 1999.

[12] Kerievsky, Joshua. Refactoring to Patterns, 2004

[13] Smells to Refactorings - Quick Reference Guide
http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

[14] Software metric
https://en.wikipedia.org/wiki/Software_metric

[15] Dr. Linda H. Rosenberg, Lawrence E. Hyatt. Software Quality Metrics for Object-Oriented Environments
http://people.ucalgary.ca/∼far/Lectures/SENG421/PDF/oocross.pdf

[16] Mesure de la qualité du code source - Algorithmes et outils
http://www-igm.univ-mlv.fr/∼dr/XPOSE2008/Mesure%20de%20la%20qualite%20du%20code%20source%20-%20Algorithmes%20et%20outils/index.html

[17] Essays, UK. (November 2018). Weighted Method Per Class Information Technology Essay.
https://www.ukessays.com/essays/information-technology/weighted-method-per-class-information-technology-essay.php?vref=1

[18] S.Smith. Static Code Analysis and Quality Metrics
https://ardalis.com/static-code-analysis-and-quality-metrics

[19] Cohesion metrics - Project Analyzer
https://www.aivosto.com/project/help/pm-oo-cohesion.html

[20] Cheikhi, Laila and Al-Qutaish, Rafa and Idri, Ali and Sellami, Asma. 2014. Chidamber and Kemerer Object-Oriented Measures : Analysis of their Design from the Metrology Perspective
https://www.researchgate.net/publication/260835125_Chidamber_and_Kemerer_Object-Oriented_Measures_Analysis_of_their_Design_from_the_Metrology_Perspective

[21] An Overview of Object-Oriented Design Metrics.
http://www.cc.uah.es/drg/b/RodHarRama00.English.pdf

[22] On the impact of code smells on the energy consumption of mobile applications
https://www.sciencedirect.com/science/article/pii/S0950584918301678

[23] Litgtweight Detection of Android-Specific Code Smells
https://dibt.unimol.it/staff/fpalomba/documents/C18.pdf

[24] An experience report on using code smells detection tools
https://www.researchgate.net/publication/224247686_An_Experience_Report_on_Using_Code_Smells_Detection_Tools

[25] JSpIRIT : a flexible tool for the analysis of code smells
https://www.researchgate.net/publication/300416283_JSpIRIT_a_flexible_tool_for_the_analysis_of_code_smells

[26] Code Smell Detecting Tool and Code Smell-Structure Bug Relationship
https://www.researchgate.net/publication/261259161_Code_Smell_Detecting_Tool_and_Code_Smell-Structure_Bug_Relationship

[27] Naouel Moha, Yann-Gal Guhneuc, Laurence Duchien et Anne-Franoise Le Meur, DECOR : A Method for the Specification and Detection of Code and Design Smells , IEEE Transactions on Software Engineering, vol. 36, no 1, janvier-fvrier 2010, p. 20-36

[28] Fabio Palomba, Rocco Oliveto, Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta et Denys Poshyvanyk, Detecting Bad Smells in Source Code using Change History Information , IEEE International Conference on Automated Software Engineering, 2013, p. 268-278

[29] Detecting Bad Smells in Source Code Using Change History Information
https://dibt.unimol.it/staff/fpalomba/documents/C2.pdf

[30] Francesca Arcelli Fontana, Pietro Braione, Marco Zanoni. Automatic detection of bad smells in code : An experimental assessment
http://www.jot.fm/issues/issue_2012_08/article5.pdf

[31] Site web de CheckStyle
http://checkstyle.sourceforge.net/

[32] J. Reimann, M. Brylski, and U. Amann, A tool-supported quality smell catalogue for android developers, 2014.

[33] A Troublesome Legacy : Memory Leaks in Java
https://dzone.com/articles/a-troublesome-legacy-memory-leaks-in-java

[34] Documentation officielle de Android : Interface Closeable
https://developer.android.com/reference/java/io/Closeable

[35] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia. Lightweight detection of android-specific code smells : the adoctor project 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2017), pp. 487-491

[36] Quantifying the Effect of Code Smells on Maintenance Effort
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.303.1555&rep=rep1&type=pdf

[37] M. Zhang, T. Hall and N. Baddoo, Code Bad Smells : A Review of Current Knowledge, J. Softw. Maint. 23(3) : pp. 179202, 2011.

[38] La production de l'électricité et ses émissions de CO2
https://www.planete-energies.com/fr/medias/decryptages/la-production-de-l-electricite-et-ses-emissions-de-co2

[39] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness
http://swat.polymtl.ca/∼foutsekh/docs/TechReport.pdf

[40] Empirical Evaluation of the Energy Impact of Refactoring Code Smells
https://research.vu.nl/ws/portalfiles/portal/47979804/ICT4S_2018.pdf

[41] G. Procaccianti, H. Fernandez, and P. Lago, Empirical Evaluation of Two Best-Practices for Energy-Efficient Software Development, J. Syst. Softw., vol. 117, no. July 2016, pp. 185198, 2016.
https://www.sciencedirect.com/science/article/pii/S0164121216000777

[42] F. Alizadeh Moghaddam, G. Procaccianti, G. A. Lewis, and P. Lago, Empirical validation of cyber-foraging architectural tactics for surrogate provisioning, The Journal of systems and software, vol. 138, no. 4, pp. 3751, Apr. 2018.
https://www.sciencedirect.com/science/article/pii/S0164121217302832

[43] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, The evolution and impact of code smells : A case study of two open source systems, in Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement. IEEE Computer Society, 2009, pp. 390400.

[44] A. Yamashita and S. Counsell, Code smells as system-level indicators of maintainability : An empirical study, Journal of Systems and Software, vol. 86, no. 10, pp. 26392653, 2013.

[45] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle.
Energy profiles of java collections classes Proceedings of the 38th International Conference on Software Engineering, ICSE 16, ACM, New York, NY, USA (2016), pp. 225-236

[46] A. Carette, M.A.A. Younes, G. Hecht, N. Moha, R. Rouvoy.
Investigating the energy impact of android smells 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2017), pp. 115-126

[47] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia.
Software-based energy profiling of android apps : Simple, efficient and reliable ? 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2017), pp. 103-114

[48] Documentation Android - PowerManager.WakeLock.
https://developer.android.com/reference/android/os/PowerManager.WakeLock

[49] Alexander Shvets. Gerhard Frey. Marina Pavlova - Refactoring techniques
https://sourcemaking.com/refactoring/refactorings

[50] Energy Efficiency across Programming Languages
http://greenlab.di.uminho.pt/wp-content/uploads/2017/09/paperSLE.pdf

[51] http://jrefactory.sourceforge.net/

[52] FaultBuster : An Automatic Code Smell Refactoring Toolset
http://www.inf.u-szeged.hu/~ncsaba/research/pdfs/2015/Szoke-SCAM2015-preprint.pdf

[53] IntelliJ. IDEA.
http://www.intellij.com/idea

[54] Eclipse IDE.
http://www.eclipse.org

[55] Taina J. How green is your software ? In : Tyrvinen P, Cusumano MA, Jansen S, editors. Software Business, First International Conference, ICSOB 2010, Jyv askyl a, Finland, June 21-23, 2010, Proceedings, Berlin, Heidelberg ; 2010. p.151-62.

[56] Influence Diagrams.
http://www.lumina.com/technology/influence-diagrams/